Tutorial - RDFLib

Ruijie Wang | 28-09-2022 Based on rdflib 6.2.0 documentation.

1. Introduction

- RDFLib is a Python package for working with knowledge graphs.
- It has been published on PyPI. We can install it via Python's package manager `pip`.

In [1]:
```
!pip install rdflib
```

```
Requirement already satisfied: rdflib in /Users/wangruijie/opt/anaconda3/lib/p
ython3.8/site-packages (6.2.0)
Requirement already satisfied: pyparsing in /Users/wangruijie/opt/anaconda3/li
b/python3.8/site-packages (from rdflib) (2.4.7)
Requirement already satisfied: isodate in /Users/wangruijie/opt/anaconda3/lib/
python3.8/site-packages (from rdflib) (0.6.1)
Requirement already satisfied: setuptools in /Users/wangruijie/opt/anaconda3/l
ib/python3.8/site-packages (from rdflib) (50.3.1.post20201107)
Requirement already satisfied: six in /Users/wangruijie/opt/anaconda3/lib/pyth
on3.8/site-packages (from isodate->rdflib) (1.15.0)
```

- RDFLib has the following features:
  - Parsers & Serializers (RDF/XML, N3, NTriples, N-Quads, Turtle, TriX, JSON-LD, RDFa and Microdata)
  - Store implementations (for in-memory and persistent knowledge graph storage)
  - Graph interface (for single graphs and datasets of multiple graphs)
  - SPARQL 1.1 implementation (both Queries and Updates are supported)

2. Questions to Consider

- How to create entities and relations?
- How to create a knowledge graph?
- How to store and load knowledge graphs?
- How to search in knowledge graphs?

3. How to create entities and relations?

- There are three classes in RDFlib that we can use to create entities and relations: URIRef, BNode, and Literal.

3.1 URIRef

- URIRef can be used to create both entities and relations that have exact URIs.

In [2]:
```
from rdflib import URIRef

# create example entities
uzh = URIRef('http://example.org/UZH')
university = URIRef('http://example.org/University')

# create an example relation
data_type = URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#type')

print(' UZH entity: {},\n university entity: {},\n data type relation: {}'.fo
```

```
UZH entity: http://example.org/UZH,
university entity: http://example.org/University,
data type relation: http://www.w3.org/1999/02/22-rdf-syntax-ns#type
```

- RDFLib supports defining namespaces when creating entities/relations with URIs.

In [3]:
```
from rdflib import Namespace

# define a namespace
EX = Namespace('http://example.org/')
```

- There are two styles of creating entities in defined namespaces.

In [4]:
```
# object attribute-like style
uzh = EX.UZH

# dictionary-like style
university = EX['University']

print(' UZH entity: {},\n university entity: {}'.format(uzh, university))
```

```
UZH entity: http://example.org/UZH,
university entity: http://example.org/University
```

- Several commonly used namespaces, such as RDF, RDFS, OWL, and FOAF, have been pre-defined in RDFLib.

In [5]:
```
from rdflib.namespace import RDF

# define a URIRef relation with a pre-defined namespace
data_type = RDF.type

print(' data type relation: {}'.format(data_type))
```

```
data type relation: http://www.w3.org/1999/02/22-rdf-syntax-ns#type
```

3.2 BNode

- BNode is used to create entities that have unknown URIs - usually entities with identity in relation to other entities.

In [6]:
```
from rdflib import BNode

# create a blank node
enrolled_students = BNode()

# balnk nodes are local identifiers for unnamed entities in knowledge graphs
print(' a blank node: {}'.format(enrolled_students))
```

```
a blank node: N44368b058dd240bd8401888a0326ec74
```

3.3 Literal

- Literals are used to create attribute values, such as a person's name, a date, a number, etc.

In [7]:
```
from rdflib import Literal

# create a literal
uzh_label = Literal('University of Zurich')

print(' uzh label: {}'.format(uzh_label))
```

```
uzh label: University of Zurich
```

- You can specify data types when creating literals.

In [8]:
```
# import XSD (XML Schema Definition) to define datatypes
from rdflib.namespace import XSD

# define an integer literal
num_students = Literal(28000, datatype=XSD.integer)

# define a strig literal with a language tag
uzh_label = Literal('University of Zurich', lang='en')

print(' number of students: {},\n uzh label: {}'.format(num_students, uzh_lab

uzh_label
```

```
number of students: 28000,
uzh label: University of Zurich
```

Out[8]:
```
rdflib.term.Literal('University of Zurich', lang='en')
```

4. How to create a knowledge graph?

- RDFLib defines the class Graph for organizing knowledge graphs.

In [9]:
```
from rdflib import Graph

# create a knowledge graph object
uzh_graph = Graph()

# check the number of triples in a knowledge graph
print(' number of triples: {}'.format(len(uzh_graph)))
```

```
number of triples: 0
```

- The function add() can be used to add triples to a knowledge graph:

In [10]:
```
# add triples to the UZH knowledge graph

uzh_graph.add((uzh, data_type, university))
uzh_graph.add((uzh, EX.has, enrolled_students))
uzh_graph.add((enrolled_students, EX.size, num_students))
uzh_graph.add((uzh, EX.label, uzh_label))

print('umber of triples: {}'.format(len(uzh_graph)))
```

```
umber of triples: 4
```

- The `for x in y:` loop can be used to loop through all triples in a knowledge graph
- A triple can be regarded as a tuple of three elements

In [11]:
```
num_trps = 0
for triple in uzh_graph:
    print(' triple-{}: ({}, {}, {})'.format(num_trps, triple[0], triple[1], t
    num_trps += 1
```

```
 triple-0: (http://example.org/UZH, http://example.org/label, University of Zu
rich)
 triple-1: (http://example.org/UZH, http://example.org/has, N44368b058dd240bd8
401888a0326ec74)
 triple-2: (http://example.org/UZH, http://www.w3.org/1999/02/22-rdf-syntax-ns
#type, http://example.org/University)
 triple-3: (N44368b058dd240bd8401888a0326ec74, http://example.org/size, 28000)
```

- Triples can be removed by the function remove():

In [12]:
```
uzh_graph.remove((uzh, EX.has, enrolled_students))

print('umber of triples: {}'.format(len(uzh_graph)))
```

```
umber of triples: 3
```

- It is possible to define a triple pattern to remove a pattern of triples.

In [13]:
```
print('before removing triples that have {} as subjects: \n'.format(uzh))
for trp_id, triple in enumerate(uzh_graph):
    print(' triple-{}: ({}, {}, {})'.format(trp_id, triple[0], triple[1], tri
```

```
before removing triples that have http://example.org/UZH as subjects:

 triple-0: (http://example.org/UZH, http://www.w3.org/1999/02/22-rdf-syntax-ns
#type, http://example.org/University)
 triple-1: (http://example.org/UZH, http://example.org/label, University of Zu
rich)
 triple-2: (N44368b058dd240bd8401888a0326ec74, http://example.org/size, 28000)
```

In [14]:
```
# remove a pattern of triples
uzh_graph.remove((uzh, None, None))

print('after removing triples that have {} as subjects: \n'.format(uzh))
for trp_id, triple in enumerate(uzh_graph):
    print(' triple-{}: ({}, {}, {})'.format(trp_id, triple[0], triple[1], tri
```

```
after removing triples that have http://example.org/UZH as subjects:

 triple-0: (N44368b058dd240bd8401888a0326ec74, http://example.org/size, 28000)
```

5. How to store and load knowledge graphs?

- The function serialize() can be used to store a knowledge graph to a file.

In [15]:
```
# resume the UZH knowledge graph
uzh_graph = Graph()
uzh_graph.add((uzh, data_type, university))
uzh_graph.add((uzh, EX.has, enrolled_students))
uzh_graph.add((enrolled_students, EX.size, num_students))
uzh_graph.add((uzh, EX.label, uzh_label))

# bind the EX namespace with the prefix "example"
uzh_graph.bind('example', EX)
```

```
# store the UZH knowledge graph to a local file in the turtle format
uzh_graph.serialize(destination='./demo.ttl', format='turtle')

# check the stored knowledge graph
!cat './demo.ttl'
```

```
@prefix example: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

example:UZH a example:University ;
    example:has [ example:size 28000 ] ;
    example:label "University of Zurich"@en .
```

In [16]:
```
# store the UZH knowledge graph to a local file in the ntriples format
uzh_graph.serialize(destination='./demo.nt', format='nt', encoding='utf-8')

# check the stored knowledge graph
!cat './demo.nt'
```

```
<http://example.org/UZH> <http://example.org/label> "University of Zurich"@en
.
<http://example.org/UZH> <http://example.org/has> _:N44368b058dd240bd8401888a0
326ec74 .
<http://example.org/UZH> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <ht
tp://example.org/University> .
_:N44368b058dd240bd8401888a0326ec74 <http://example.org/size> "28000"^^<htt
p://www.w3.org/2001/XMLSchema#integer> .
```

- serialize() can also be used to print a knowledge graph object

In [17]:
```
print(uzh_graph.serialize(format='nt'))
```

```
<http://example.org/UZH> <http://example.org/label> "University of Zurich"@en
.
<http://example.org/UZH> <http://example.org/has> _:N44368b058dd240bd8401888a0
326ec74 .
<http://example.org/UZH> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <ht
tp://example.org/University> .
_:N44368b058dd240bd8401888a0326ec74 <http://example.org/size> "28000"^^<htt
p://www.w3.org/2001/XMLSchema#integer> .
```

- The function parse() can be used to load knowledge graphs

In [18]:
```
# define an empty knowledge graph
uzh_graph = Graph()

# load a knowledge graph
uzh_graph.parse(source='./demo.nt', format='nt')

for trp_id, triple in enumerate(uzh_graph):
    print(' triple-{}: ({}, {}, {})'.format(trp_id, triple[0], triple[1], tri
```

```
 triple-0: (Nc77e3eff41684bd7b241f5a8251ad068, http://example.org/size, 28000)
 triple-1: (http://example.org/UZH, http://example.org/has, Nc77e3eff41684bd7b
241f5a8251ad068)
 triple-2: (http://example.org/UZH, http://example.org/label, University of Zu
rich)
 triple-3: (http://example.org/UZH, http://www.w3.org/1999/02/22-rdf-syntax-ns
#type, http://example.org/University)
```

- Rdflib can guess the file format by the file's name. For example, ".nt" is commonly used for n-triple files. We can use `parse()` to load a ".nt" file without specifiying the file format.

In [19]:
```
# define an empty knowledge graph
uzh_graph = Graph()

# load a knowledge graph
uzh_graph.parse(source='./demo.nt')

for trp_id, triple in enumerate(uzh_graph):
    print(' triple-{}: ({}, {}, {})'.format(trp_id, triple[0], triple[1], tri
```

```
 triple-0: (N69452b7c36c5474caec2b48617ee3440, http://example.org/size, 28000)
 triple-1: (http://example.org/UZH, http://www.w3.org/1999/02/22-rdf-syntax-ns
#type, http://example.org/University)
 triple-2: (http://example.org/UZH, http://example.org/has, N69452b7c36c5474ca
ec2b48617ee3440)
 triple-3: (http://example.org/UZH, http://example.org/label, University of Zu
rich)
```

6. How to search in knowledge graphs?

- RDFLib's Graph objects support "containing" check.

In [20]:
```
# check if a specific triple exists
triple = (uzh, RDF.type, university)
if triple in uzh_graph:
    print(' the triple {} exists \n'.format(triple))
else:
    print(' the triple {} does not exist \n'.format(triple))


# check if a pattern of triples exist
triple_pattern = (None, RDF.type, None)
if triple_pattern in uzh_graph:
    print(' there is triple like {}'.format(triple_pattern))
else:
    print(' there is no triple like {}'.format(triple_pattern))
```

```
 the triple (rdflib.term.URIRef('http://example.org/UZH'), rdflib.term.URIRef
('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'), rdflib.term.URIRef('htt
p://example.org/University')) exists

 there is triple like (None, rdflib.term.URIRef('http://www.w3.org/1999/02/22-
rdf-syntax-ns#type'), None)
```

- RDFLib Graph objects support triple pattern matching with the triples() function.
- The function triples() returns a generator of matched triples.

In [21]:
```
# define a triple pattern
triple_pattern = (uzh, None, None)

# search all triples that match the defined pattern
triple_generator = uzh_graph.triples(triple_pattern)
for triple in triple_generator:
    print(triple)
```

```
(rdflib.term.URIRef('http://example.org/UZH'), rdflib.term.URIRef('http://exam
ple.org/label'), rdflib.term.Literal('University of Zurich', lang='en'))
(rdflib.term.URIRef('http://example.org/UZH'), rdflib.term.URIRef('http://exam
ple.org/has'), rdflib.term.BNode('N69452b7c36c5474caec2b48617ee3440'))
(rdflib.term.URIRef('http://example.org/UZH'), rdflib.term.URIRef('http://www.
w3.org/1999/02/22-rdf-syntax-ns#type'), rdflib.term.URIRef('http://example.or
g/University'))
```

- If you are not interested in whole triples, you can use subjects(), predicates(), and

objects() to retrieve only subjects, predicates, and objects.

In [22]:
```
# retrieve subjects that have the label "University of Zurich"
for subjs in uzh_graph.subjects(EX.label, uzh_label):
    print(subjs)
```

```
http://example.org/UZH
```

In [23]:
```
# retrieve predicates between any two entities
for preds in uzh_graph.predicates(None, None):
    print(preds)
```

```
http://example.org/size
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://example.org/has
http://example.org/label
```

In [24]:
```
# retrieve all objects of the entity uzh
for objs in uzh_graph.objects(uzh, None):
    print(objs)
```

```
University of Zurich
N69452b7c36c5474caec2b48617ee3440
http://example.org/University
```

- subjects(), predicates(), and objects() can also be used to retrieve all subjects, predicates, and objects in a knowledge graph when no argument is given

In [25]:
```
print(' all subjects in the UZH knowledge graph')
for objs in set(uzh_graph.subjects()):
    print(objs)

print('\n all predicates in the UZH knowledge graph')
for objs in set(uzh_graph.predicates()):
    print(objs)

print('\n all objects in the UZH knowledge graph')
for objs in set(uzh_graph.objects()):
    print(objs)
```

```
 all subjects in the UZH knowledge graph
http://example.org/UZH
N69452b7c36c5474caec2b48617ee3440

 all predicates in the UZH knowledge graph
http://example.org/label
http://example.org/size
http://example.org/has
http://www.w3.org/1999/02/22-rdf-syntax-ns#type

 all objects in the UZH knowledge graph
N69452b7c36c5474caec2b48617ee3440
University of Zurich
28000
http://example.org/University
```

- RDFLib also supports querying Graph objects with SPARQL queries.

In [26]:
```
# first check prefixes
uzh_graph.bind('example', EX)
for namespace in uzh_graph.namespaces():
    print(namespace)
```

```
('owl', rdflib.term.URIRef('http://www.w3.org/2002/07/owl#'))
('rdf', rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#'))
```

```
('rdfs', rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#'))
('xsd', rdflib.term.URIRef('http://www.w3.org/2001/XMLSchema#'))
('xml', rdflib.term.URIRef('http://www.w3.org/XML/1998/namespace'))
('example', rdflib.term.URIRef('http://example.org/'))
```

In [27]:

```
# define a SPARQL query
query = '''
SELECT ?x
WHERE {
?x rdf:type example:University.
}
'''

# check the result of the SPARQL query
res = uzh_graph.query(query)
for _ in res:
    print(_)
```

```
(rdflib.term.URIRef('http://example.org/UZH'),)
```

In [ ]: