rdflib Documentation

Release 5.0.0

RDFLib Team

May 26, 2020

CONTENTS

RDFLib is a pure Python package for working with RDF. RDFLib contains useful APIs for working with RDF, including:

- **Parsers & Serializers**
  - for RDF/XML, N3, NTriples, N-Quads, Turtle, TriX, RDFa and Microdata
  - and JSON-LD, via a plugin module
- **Store implementations**
  - for in-memory and persistent RDF storage - Berkeley DB
- **Graph interface**
  - to a single graph
  - or a conjunctive graph (multiple Named Graphs)
  - or a dataset of graphs
- **SPARQL 1.1 implementation**
  - supporting both Queries and Updates

CHAPTER

ONE

GETTING STARTED

If you have never used RDFLib, the following will help get you started:

1.1 Getting started with RDFLib

1.1.1 Installation

RDFLib is open source and is maintained in a GitHub repository. RDFLib releases, current and previous are listed on PyPi

The best way to install RDFLib is to use `pip` (sudo as required):

```
$ pip install rdflib
```

If you want the latest code to run, clone the master branch of the GitHub repo and use that.

1.1.2 Support

Usage support is available via questions tagged with `[rdflib]` on StackOverflow and development support, notifications and detailed discussion through the rdflib-dev group (mailing list):

> http://groups.google.com/group/rdflib-dev

If you notice an bug or want to request an enhancement, please do so via our Issue Tracker in Github:

> http://github.com/RDFLib/rdflib/issues

1.1.3 How it all works

The package uses various Python idioms that offer an appropriate way to introduce RDF to a Python programmer who hasn't worked with RDF before.

The primary interface that RDFLib exposes for working with RDF is a *`Graph`*.

RDFLib graphs are not sorted containers; they have ordinary `set` operations (e.g. *`add()`* to add a triple) plus methods that search triples and return them in arbitrary order.

RDFLib graphs also redefine certain built-in Python methods in order to behave in a predictable way; they emulate container types and are best thought of as a set of 3-item tuples ("triples", in RDF-speak):

```
[
    (subject0, predicate0, object0),
    (subject1, predicate1, object1),
    ...
    (subjectN, predicateN, objectN)
 ]
```

A tiny usage example:

```
import rdflib

# create a Graph
g = rdflib.Graph()

# parse in an RDF file hosted on the Internet
result = g.parse("http://www.w3.org/People/Berners-Lee/card")

# loop through each triple in the graph (subj, pred, obj)
for subj, pred, obj in g:
    # check if there is at least one triple in the Graph
    if (subj, pred, obj) not in g:
       raise Exception("It better be!")

# print the number of "triples" in the Graph
print("graph has {} statements.".format(len(g)))
# prints graph has 86 statements.

# print out the entire Graph in the RDF Turtle format
print(g.serialize(format="turtle").decode("utf-8"))
```

Here a *Graph* is created and then an RDF file online, Tim Berners-Lee's social network details, is parsed into that graph. The `print()` statement uses the `len()` function to count the number of triples in the graph.

A more extensive example:

```
from rdflib import Graph, Literal, RDF, URIRef
# rdflib knows about some namespaces, like FOAF
from rdflib.namespace import FOAF , XSD

# create a Graph
g = Graph()

# Create an RDF URI node to use as the subject for multiple triples
donna = URIRef("http://example.org/donna")

# Add triples using store's add() method.
g.add((donna, RDF.type, FOAF.Person))
g.add((donna, FOAF.nick, Literal("donna", lang="ed")))
g.add((donna, FOAF.name, Literal("Donna Fales")))
g.add((donna, FOAF.mbox, URIRef("mailto:donna@example.org")))

# Add another person
ed = URIRef("http://example.org/edward")

# Add triples using store's add() method.
g.add((ed, RDF.type, FOAF.Person))
g.add((ed, FOAF.nick, Literal("ed", datatype=XSD.string)))
g.add((ed, FOAF.name, Literal("Edward Scissorhands")))
```

(continues on next page)

(continued from previous page)

```
g.add((ed, FOAF.mbox, URIRef("mailto:e.scissorhands@example.org")))

# Iterate over triples in store and print them out.
print("--- printing raw triples ---")
for s, p, o in g:
    print((s, p, o))

# For each foaf:Person in the store, print out their mbox property's value.
print("--- printing mboxes ---")
for person in g.subjects(RDF.type, FOAF.Person):
    for mbox in g.objects(person, FOAF.mbox):
        print(mbox)

# Bind the FOAF namespace to a prefix for more readable output
g.bind("foaf", FOAF)

# print all the data in the Notation3 format
print("--- printing mboxes ---")
print(g.serialize(format='n3').decode("utf-8"))
```

1.1.4 More examples

There are many more *examples* in the `examples` folder in the source distribution.

1.2 Loading and saving RDF

1.2.1 Reading an n-triples file

RDF data has various syntaxes (`xml`, `n3`, `ntriples`, `trix`, `JSON-LD`, etc) that you might want to read. The simplest format is `ntriples`, a line-based format. Create the file `demo.nt` in the current directory with these two lines:

```
<http://bigasterisk.com/foaf.rdf#drewp> <http://www.w3.org/1999/02/22-rdf-syntax-ns
↪#type> <http://xmlns.com/foaf/0.1/Person> .
<http://bigasterisk.com/foaf.rdf#drewp> <http://example.com/says> "Hello world" .
```

You need to tell RDFLib what format to parse, use the `format` keyword-parameter to *`parse()`*, you can pass either a mime-type or the name (a *list of available parsers* is available). If you are not sure what format your file will be, you can use *`rdflib.util.guess_format()`* which will guess based on the file extension.

In an interactive python interpreter, try this:

```
from rdflib import Graph

g = Graph()
g.parse("demo.nt", format="nt")

print(len(g)) # prints 2

import pprint
for stmt in g:
    pprint.pprint(stmt)
```

(continues on next page)

(continued from previous page)

```
# prints :
(rdflib.term.URIRef('http://bigasterisk.com/foaf.rdf#drewp'),
 rdflib.term.URIRef('http://example.com/says'),
 rdflib.term.Literal('Hello world'))
(rdflib.term.URIRef('http://bigasterisk.com/foaf.rdf#drewp'),
 rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'),
 rdflib.term.URIRef('http://xmlns.com/foaf/0.1/Person'))
```

The final lines show how RDFLib represents the two statements in the file. The statements themselves are just length-3 tuples; and the subjects, predicates, and objects are all rdflib types.

1.2.2 Reading remote graphs

Reading graphs from the net is just as easy:

```
g.parse("http://bigasterisk.com/foaf.rdf")
print(len(g))
# prints 42
```

The format defaults to `xml`, which is the common format for .rdf files you'll find on the net.

RDFLib will also happily read RDF from any file-like object, i.e. anything with a `.read()` method.

1.3 Creating RDF triples

1.3.1 Creating Nodes

RDF data is a graph where the nodes are URI references, Blank Nodes or Literals. In RDFLib, these node types are represented by the classes *URIRef*, *BNode*, and *Literal*. `URIRefs` and `BNodes` can both be thought of as resources, such a person, a company, a website, etc.

- A `BNode` is a node where the exact URI is not known.
- A `URIRef` is a node where the exact URI is knonw. `URIRefs` are also used to represent the properties/predicates in the RDF graph.
- `Literals` represent attribute values, such as a name, a date, a number, etc. The most common literal values are XML data types, e.g. string, int...

Nodes can be created by the constructors of the node classes:

```
from rdflib import URIRef, BNode, Literal

bob = URIRef("http://example.org/people/Bob")
linda = BNode()  # a GUID is generated

name = Literal('Bob')  # passing a string
age = Literal(24)  # passing a python int
height = Literal(76.5)  # passing a python float
```

Literals can be created from Python objects, this creates `data-typed literals`, for the details on the mapping see *Literals*.

For creating many `URIRefs` in the same `namespace`, i.e. URIs with the same prefix, RDFLib has the *rdflib.namespace.Namespace* class:

```
from rdflib import Namespace

n = Namespace("http://example.org/people/")

n.bob  # = rdflib.term.URIRef(u'http://example.org/people/bob')
n.eve  # = rdflib.term.URIRef(u'http://example.org/people/eve')
```

This is very useful for schemas where all properties and classes have the same URI prefix. RDFLib defines Namespaces for some common RDF/OWL schemas, including most W3C ones:

```
from rdflib.namespace import CSVW, DC, DCAT, DCTERMS, DOAP, FOAF, ODRL2, ORG, OWL, \
                             PROF, PROV, RDF, RDFS, SDO, SH, SKOS, SOSA, SSN, TIME, \
                             VOID, XMLNS, XSD

RDF.type
# = rdflib.term.URIRef("http://www.w3.org/1999/02/22-rdf-syntax-ns#type")

FOAF.knows
# = rdflib.term.URIRef("http://xmlns.com/foaf/0.1/knows")

PROF.isProfileOf
# = rdflib.term.URIRef("http://www.w3.org/ns/dx/prof/isProfileOf")

SOSA.Sensor
# = rdflib.term.URIRef("http://www.w3.org/ns/sosa/Sensor")
```

1.3.2 Adding Triples

We already saw in *Loading and saving RDF*, how triples can be added from files and online locations with with the *parse()* function.

Triples can also be added within Python code directly, using the *add()* function:

`Graph.`**`add`**`(`*triple*`)`
Add a triple with self as context

add() takes a 3-tuple (a "triple") of RDFLib nodes. Try the following with the nodes and namespaces we defined previously:

```
from rdflib import Graph
g = Graph()
g.bind("foaf", FOAF)

g.add((bob, RDF.type, FOAF.Person))
g.add((bob, FOAF.name, name))
g.add((bob, FOAF.knows, linda))
g.add((linda, RDF.type, FOAF.Person))
g.add((linda, FOAF.name, Literal("Linda")))

print(g.serialize(format="turtle").decode("utf-8"))
```

outputs:

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<http://example.org/people/Bob> a foaf:Person ;
    foaf:knows [ a foaf:Person ;
            foaf:name "Linda" ] ;
    foaf:name "Bob" .
```

For some properties, only one value per resource makes sense (i.e they are *functional properties*, or have max-cardinality of 1). The *set()* method is useful for this:

```
g.add((bob, FOAF.age, Literal(42)))
print("Bob is ", g.value(bob, FOAF.age))
# prints: Bob is 42

g.set((bob, FOAF.age, Literal(43)))  # replaces 42 set above
print("Bob is now ", g.value(bob, FOAF.age))
# prints: Bob is now 43
```

rdflib.graph.Graph.value() is the matching query method, it will return a single value for a property, optionally raising an exception if there are more.

You can also add triples by combining entire graphs, see *Set Operations on RDFLib Graphs*.

1.3.3 Removing Triples

Similarly, triples can be removed by a call to *remove()*:

Graph.**remove**(*triple*)
Remove a triple from the graph

If the triple does not provide a context attribute, removes the triple from all contexts.

When removing, it is possible to leave parts of the triple unspecified (i.e. passing `None`), this will remove all matching triples:

```
g.remove((bob, None, None))  # remove all triples about bob
```

1.3.4 An example

LiveJournal produces FOAF data for their users, but they seem to use `foaf:member_name` for a person's full name but `foaf:member_name` isn't in FOAF's namespace and perhaps they should have used `foaf:name`

To retrieve some LiveJournal data, add a `foaf:name` for every `foaf:member_name` and then remove the `foaf:member_name` values to ensure the data actually aligns with other FOAF data, we could do this:

```
from rdflib import Graph
from rdflib.namespace import FOAF

g = Graph()
# get the data
g.parse("http://danbri.livejournal.com/data/foaf")

# for every foaf:member_name, add foaf:name and remove foaf:member_name
for s, p, o in g.triples((None, FOAF['member_name'], None)):
    g.add((s, FOAF['name'], o))
    g.remove((s, FOAF['member_name'], o))
```

Note: Since rdflib 5.0.0, using `foaf:member_name` is somewhat prevented in RDFlib since FOAF is declared as a *ClosedNamespace()* class instance that has a closed set of members and `foaf:member_name` isn't one of them! If LiveJournal used RDFlib 5.0.0, an error would have been raised for `foaf:member_name` when the triple was created.

1.3.5 Creating Containers & Collections

There are two convenience classes for RDF Containers & Collections which you can use instead of declaring each triple of a Containers or a Collections individually:

- *Container()* (also `Bag`, `Seq` & `Alt`) and
- *Collection()*

See their documentation for how.

1.4 Navigating Graphs

An RDF Graph is a set of RDF triples, and we try to mirror exactly this in RDFLib. The Python *Graph()* tries to emulate a container type.

1.4.1 Graphs as Iterators

RDFLib graphs override *__iter__()* in order to support iteration over the contained triples:

```
for subject, predicate, object in someGraph:
    if not (subject, predicate, object) in someGraph:
        raise Exception("Iterator / Container Protocols are Broken!!")
```

1.4.2 Contains check

Graphs implement *__contains__()*, so you can check if a triple is in a graph with `triple in graph` syntax:

```
from rdflib import URIRef
from rdflib.namespace import RDF
bob = URIRef("http://example.org/people/bob")
if (bob, RDF.type, FOAF.Person) in graph:
    print("This graph knows that Bob is a person!")
```

Note that this triple does not have to be completely bound:

```
if (bob, None, None) in graph:
    print("This graph contains triples about Bob!")
```

1.4.3 Set Operations on RDFLib Graphs

Graphs override several pythons operators: *__iadd__()*, *__isub__()*, etc. This supports addition, subtraction and other set-operations on Graphs:

operation	effect
`G1 + G2`	return new graph with union
`G1 += G1`	in place union / addition
`G1 - G2`	return new graph with difference
`G1 -= G2`	in place difference / subtraction
`G1 & G2`	intersection (triples in both graphs)
`G1 ^ G2`	xor (triples in either G1 or G2, but not in both)

Warning: Set-operations on graphs assume Blank Nodes are shared between graphs. This may or may not do what you want. See *Merging graphs* for details.

1.4.4 Basic Triple Matching

Instead of iterating through all triples, RDFLib graphs support basic triple pattern matching with a *triples()* function. This function is a generator of triples that match the pattern given by the arguments. The arguments of these are RDF terms that restrict the triples that are returned. Terms that are `None` are treated as a wildcard. For example:

```
g.load("some_foaf.rdf")
for s, p, o in g.triples((None, RDF.type, FOAF.Person)):
    print("{} is a person".format(s))

for s, p, o in g.triples((None,  RDF.type, None)):
    print("{} is a {}".format(s, o))

bobgraph = Graph()

bobgraph += g.triples((bob, None, None))
```

If you are not interested in whole triples, you can get only the bits you want with the methods *objects()*, *subjects()*, *predicates()*, *predicate_objects()*, etc. Each take parameters for the components of the triple to constraint:

```
for person in g.subjects(RDF.type, FOAF.Person):
    print("{} is a person".format(person))
```

Finally, for some properties, only one value per resource makes sense (i.e they are *functional properties*, or have max-cardinality of 1). The *value()* method is useful for this, as it returns just a single node, not a generator:

```
name = g.value(bob, FOAF.name) # get any name of bob
# get the one person that knows bob and raise an exception if more are found
mbox = g.value(predicate = FOAF.name, object=bob, any=False)
```

1.4.5 `Graph` methods for accessing triples

Here is a list of all convenience methods for querying Graphs:

`Graph.`**`label`**(*subject*, *default=''*)
Query for the RDFS.label of the subject

Return default if no label exists or any label if multiple exist.

`Graph.`**`preferredLabel`**(*subject*, *lang=None*, *default=None*, *labelProperties=rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'), rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#label')*)
Find the preferred label for subject.

By default prefers skos:prefLabels over rdfs:labels. In case at least one prefLabel is found returns those, else returns labels. In case a language string (e.g., "en", "de" or even "" for no lang-tagged literals) is given, only such labels will be considered.

Return a list of (labelProp, label) pairs, where labelProp is either skos:prefLabel or rdfs:label.

```
>>> from rdflib import ConjunctiveGraph, URIRef, RDFS, Literal
>>> from rdflib.namespace import SKOS
>>> from pprint import pprint
>>> g = ConjunctiveGraph()
>>> u = URIRef("http://example.com/foo")
>>> g.add([u, RDFS.label, Literal("foo")])
>>> g.add([u, RDFS.label, Literal("bar")])
>>> pprint(sorted(g.preferredLabel(u)))
[(rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#label'),
  rdflib.term.Literal('bar')),
 (rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#label'),
  rdflib.term.Literal('foo'))]
>>> g.add([u, SKOS.prefLabel, Literal("bla")])
>>> pprint(g.preferredLabel(u))
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('bla'))]
>>> g.add([u, SKOS.prefLabel, Literal("blubb", lang="en")])
>>> sorted(g.preferredLabel(u))
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('bla')),
  (rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('blubb', lang='en'))]
>>> g.preferredLabel(u, lang="")
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('bla'))]
>>> pprint(g.preferredLabel(u, lang="en"))
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('blubb', lang='en'))]
```

`Graph.`**`triples`**(*triple*)
Generator over the triple store

Returns triples that match the given triple pattern. If triple pattern does not provide a context, all contexts will be searched.

`Graph.`**`value`**(*subject=None*, *predicate=rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#value')*, *object=None*, *default=None*, *any=True*)
Get a value for a pair of two criteria

Exactly one of subject, predicate, object must be None. Useful if one knows that there may only be one value.

It is one of those situations that occur a lot, hence this 'macro' like utility

Parameters: subject, predicate, object – exactly one must be None default – value to be returned if no values found any – if True, return any value in the case there is more than one, else, raise UniquenessError

`Graph.`**`subjects`**(*predicate=None*, *object=None*)
A generator of subjects with the given predicate and object

`Graph.`**`objects`**(*subject=None*, *predicate=None*)
A generator of objects with the given subject and predicate

`Graph.`**`predicates`**(*subject=None*, *object=None*)
A generator of predicates with the given subject and object

`Graph.`**`subject_objects`**(*predicate=None*)
A generator of (subject, object) tuples for the given predicate

`Graph.`**`subject_predicates`**(*object=None*)
A generator of (subject, predicate) tuples for the given object

`Graph.`**`predicate_objects`**(*subject=None*)
A generator of (predicate, object) tuples for the given subject

1.5 Querying with SPARQL

1.5.1 Run a Query

The RDFLib comes with an implementation of the SPARQL 1.1 Query and SPARQL 1.1 Update languages.

Queries can be evaluated against a graph with the *`rdflib.graph.Graph.query()`* method, and updates with *`rdflib.graph.Graph.update()`*.

The query method returns a *`rdflib.query.Result`* instance. For SELECT queries, iterating over this return `rdflib.query.ResultRow` instances, each containing a set of variable bindings. For CONSTRUCT/DESCRIBE queries, iterating over the result object gives the triples. For ASK queries, iterating will yield the single boolean answer, or evaluating the result object in a boolean-context (i.e. `bool(result)`)

Continuing the example...

```
import rdflib

g = rdflib.Graph()

# ... add some triples to g somehow ...
g.parse("some_foaf_file.rdf")

qres = g.query(
    """SELECT DISTINCT ?aname ?bname
       WHERE {
          ?a foaf:knows ?b .
          ?a foaf:name ?aname .
          ?b foaf:name ?bname .
       }""")

for row in qres:
    print("%s knows %s" % row)
```

The results are tuples of values in the same order as your SELECT arguments. Alternatively, the values can be accessed by variable name, either as attributes, or as items: `row.b` and `row["b"]` is equivalent.

```
Timothy Berners-Lee knows Edd Dumbill
Timothy Berners-Lee knows Jennifer Golbeck
Timothy Berners-Lee knows Nicholas Gibbins
Timothy Berners-Lee knows Nigel Shadbolt
Dan Brickley knows binzac
Timothy Berners-Lee knows Eric Miller
Drew Perttula knows David McClosky
Timothy Berners-Lee knows Dan Connolly
...
```

As an alternative to using `PREFIX` in the SPARQL query, namespace bindings can be passed in with the `initNs` kwarg, see *Namespaces and Bindings*.

Variables can also be pre-bound, using `initBindings` kwarg can be used to pass in a `dict` of initial bindings, this is particularly useful for prepared queries, as described below.

1.5.2 Query a Remote Service

The SERVICE keyword of SPARQL 1.1 can send a query to a remote SPARQL endpoint.

```
import rdflib

g = rdflib.Graph()
qres = g.query('''
SELECT ?s
WHERE {
  SERVICE <http://dbpedia.org/sparql> {
    ?s <http://purl.org/linguistics/gold/hypernym> <http://dbpedia.org/resource/
↪Leveller> .
  }
} LIMIT 3''')
for row in qres:
    print(row.s)
```

This example sends a query to DBPedia's SPARQL endpoint service so that it can run the query and then send back the result:

```
http://dbpedia.org/resource/Elizabeth_Lilburne
http://dbpedia.org/resource/Thomas_Prince_(Leveller)
http://dbpedia.org/resource/John_Lilburne
```

1.5.3 Prepared Queries

RDFLib lets you *prepare* queries before execution, this saves re-parsing and translating the query into SPARQL Algebra each time.

The method `rdflib.plugins.sparql.prepareQuery()` takes a query as a string and will return a *`rdflib.plugins.sparql.sparql.Query`* object. This can then be passed to the *`rdflib.graph.Graph.query()`* method.

The `initBindings` kwarg can be used to pass in a `dict` of initial bindings:

```
q = prepareQuery(
        'SELECT ?s WHERE { ?person foaf:knows ?s .}',
        initNs = { "foaf": FOAF })
```

(continues on next page)

(continued from previous page)

```
g = rdflib.Graph()
g.load("foaf.rdf")

tim = rdflib.URIRef("http://www.w3.org/People/Berners-Lee/card#i")

for row in g.query(q, initBindings={'person': tim}):
        print row
```

1.5.4 Custom Evaluation Functions

For experts, it is possible to override how bits of SPARQL algebra are evaluated. By using the setuptools entry-point `rdf.plugins.sparqleval`, or simply adding to an entry to *`rdflib.plugins.sparql.CUSTOM_EVALS`*, a custom function can be registered. The function will be called for each algebra component and may raise `NotImplementedError` to indicate that this part should be handled by the default implementation.

See `examples/custom_eval.py`

1.6 Utilities and convenience functions

For RDF programming, RDFLib and Python may not execute the fastest, but we try hard to make it the fastest and most convenient way to write!

This is a collection of hints and pointers for hassle free RDF-coding.

1.6.1 User-friendly labels

Use *`label()`* to quickly look up the RDFS label of something, or better use *`preferredLabel()`* to find a label using several different properties (i.e. either `rdfs:label`, `skos:preferredLabel`, `dc:title`, etc.).

1.6.2 Functional properties

Use *`value()`* and *`set()`* to work with functional properties, i.e. properties than can only occur once for a resource.

1.6.3 Slicing graphs

Python allows slicing arrays with a `slice` object, a triple of `start`, `stop` index and step-size:

```
>>> range(10)[2:9:3]
[2, 5, 8]
```

RDFLib graphs override `__getitem__` and we pervert the slice triple to be a RDF triple instead. This lets slice syntax be a shortcut for *`triples()`*, *`subject_predicates()`*, `contains()`, and other Graph query-methods:

```
graph[:]
# same as
iter(graph)

graph[bob]
```

(continues on next page)

(continued from previous page)

```
# same as
graph.predicate_objects(bob)

graph[bob : FOAF.knows]
# same as
graph.objects(bob, FOAF.knows)

graph[bob : FOAF.knows : bill]
# same as
(bob, FOAF.knows, bill) in graph

graph[:FOAF.knows]
# same as
graph.subject_objects(FOAF.knows)

...
```

See *examples.slice* for a complete example.

Note: Slicing is convenient for run-once scripts of playing around in the Python `REPL`. However, since slicing returns tuples of varying length depending on which parts of the slice are bound, you should be careful using it in more complicated programs. If you pass in variables, and they are `None` or `False`, you may suddenly get a generator of different length tuples back than you expect.

1.6.4 SPARQL Paths

SPARQL property paths are possible using overridden operators on URIRefs. See *examples.foafpaths* and *rdflib.paths*.

1.6.5 Serializing a single term to N3

For simple output, or simple serialisation, you often want a nice readable representation of a term. All terms have a `.n3(namespace_manager = None)` method, which will return a suitable N3 format:

```
>>> from rdflib import Graph, URIRef, Literal, BNode
>>> from rdflib.namespace import FOAF, NamespaceManager

>>> person = URIRef('http://xmlns.com/foaf/0.1/Person')
>>> person.n3()
u'<http://xmlns.com/foaf/0.1/Person>'

>>> g = Graph()
>>> g.bind("foaf", FOAF)

>>> person.n3(g.namespace_manager)
u'foaf:Person'

>>> l = Literal(2)
>>> l.n3()
u'"2"^^<http://www.w3.org/2001/XMLSchema#integer>'
```

(continues on next page)

(continued from previous page)

```
>>> l.n3(g.namespace_manager)
u'"2"^^xsd:integer'
```

1.6.6 Parsing data from a string

You can parse data from a string with the `data` param:

```
graph.parse(data = '<urn:a> <urn:p> <urn:b>.', format='n3')
```

1.6.7 Commandline-tools

RDFLib includes a handful of commandline tools, see *rdflib.tools*.

1.7 examples Package

These examples all live in `./examples` in the source-distribution of RDFLib.

1.7.1 `conjunctive_graphs` Module

An RDFLib ConjunctiveGraph is an (unnamed) aggregation of all the named graphs within a Store. The *get_context()* method can be used to get a particular named graph for use such as to add triples to, or the default graph can be used

This example shows how to create named graphs and work with the conjunction (union) of all the graphs.

1.7.2 `custom_datatype` Module

RDFLib can map between RDF data-typed literals and Python objects.

Mapping for integers, floats, dateTimes, etc. are already added, but you can also add your own.

This example shows how *rdflib.term.bind()* lets you register new mappings between literal datatypes and Python objects

1.7.3 `custom_eval` Module

This example shows how a custom evaluation function can be added to handle certain SPARQL Algebra elements.

A custom function is added that adds `rdfs:subClassOf` "inference" when asking for `rdf:type` triples.

Here the custom eval function is added manually, normally you would use setuptools and entry_points to do it: i.e. in your setup.py:

```
entry_points = {
    'rdf.plugins.sparqleval': [
        'myfunc =     mypackage:MyFunction',
        ],
}
```

examples.custom_eval.**customEval**(*ctx*, *part*)
Rewrite triple patterns to get super-classes

1.7.4 film Module

film.py: a simple tool to manage your movies review Simon Rozet, http://atonie.org/

- manage directors and writers
- manage actors
- handle non IMDB uri
- markdown support in comment

Requires download and import of Python imdb library from https://imdbpy.github.io/ - (warning: installation will trigger automatic installation of several other packages)

– Usage:

film.py whoami "John Doe <john@doe.org>" Initialize the store and set your name and email.

film.py whoami Tell you who you are

film.py http://www.imdb.com/title/tt0105236/ Review the movie "Reservoir Dogs"

class examples.film.**Store**
Bases: object

__dict__ = mappingproxy({'__module__': 'examples.film', '__init__': <function Store.

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'examples.film'

__weakref__
list of weak references to the object (if defined)

movie_is_in(*uri*)

new_movie(*movie*)

new_review(*movie*, *date*, *rating*, *comment=None*)

save()

who(*who=None*)

examples.film.**help**()

examples.film.**main**(*argv=None*)

1.7.5 `foafpaths` Module

SPARQL 1.1 defines path operators for combining/repeating predicates in triple-patterns.

We overload some Python operators on URIRefs to allow creating path operators directly in Python.

Operator	Path
`p1 / p2`	Path sequence
`p1 | p2`	Path alternative
`p1 * '*'`	chain of 0 or more p's
`p1 * '+'`	chain of 1 or more p's
`p1 * '?'`	0 or 1 p
`~p1`	p1 inverted, i.e. (s p1 o) <=> (o ~p1 s)
`-p1`	NOT p1, i.e. any property but p1

These can then be used in property position for `s,p,o` triple queries for any graph method.

See the docs for *`rdflib.paths`* for the details.

This example shows how to get the name of friends with a single query.

1.7.6 `prepared_query` Module

SPARQL Queries be prepared (i.e parsed and translated to SPARQL algebra) by the `rdflib.plugins.sparql.prepareQuery()` method.

When executing, variables can be bound with the `initBindings` keyword parameter

1.7.7 `resource` Module

RDFLib has a *`Resource`* class, for a resource-centric API.

A resource acts like a URIRef with an associated graph, and allows quickly adding or querying for triples where this resource is the subject.

1.7.8 `rdfa_example` Module

A simple example showing how to process RDFa from the web

1.7.9 `simple_example` Module

1.7.10 `sleepycat_example` Module

A simple example showing how to use a Sleepycat store to do on-disk persistence.

1.7.11 slice Module

RDFLib Graphs (and Resources) can be "sliced" with [] syntax

This is a short-hand for iterating over triples

Combined with SPARQL paths (see `foafpaths.py`) - quite complex queries can be realised.

See *rdflib.graph.Graph.__getitem__()* for details

1.7.12 smushing Module

A FOAF smushing example.

Filter a graph by normalizing all `foaf:Persons` into URIs based on their `mbox_sha1sum`.

Suppose I get two FOAF documents each talking about the same person (according to `mbox_sha1sum`) but they each used a *rdflib.term.BNode* for the subject. For this demo I've combined those two documents into one file:

This filters a graph by changing every subject with a `foaf:mbox_sha1sum` into a new subject whose URI is based on the `sha1sum`. This new graph might be easier to do some operations on.

An advantage of this approach over other methods for collapsing BNodes is that I can incrementally process new FOAF documents as they come in without having to access my ever-growing archive. Even if another `65b983bb397fb71849da910996741752ace8369b` document comes in next year, I would still give it the same stable subject URI that merges with my existing data.

1.7.13 sparql_query_example Module

SPARQL Query using *rdflib.graph.Graph.query()*

The method returns a *Result*, iterating over this yields `ResultRow` objects

The variable bindings can be access as attributes of the row objects For variable names that are not valid python identifiers, dict access (i.e. with `row[var] / __getitem__`) is also possible.

`vars` contains the variables

1.7.14 sparql_update_example Module

SPARQL Update statements can be applied with *rdflib.graph.Graph.update()*

1.7.15 sparqlstore_example Module

A simple example showing how to use the SPARQLStore

1.7.16 `swap_primer` Module

This is a simple primer using some of the example stuff in the Primer on N3:

http://www.w3.org/2000/10/swap/Primer

1.7.17 `transitive` Module

An example illustrating how to use the *`transitive_subjects()`* and *`transitive_objects()`* graph methods

Formal definition

The *`transitive_objects()`* method finds all nodes such that there is a path from subject to one of those nodes using only the predicate property in the triples. The *`transitive_subjects()`* method is similar; it finds all nodes such that there is a path from the node to the object using only the predicate property.

Informal description, with an example

In brief, *`transitive_objects()`* walks forward in a graph using a particular property, and *`transitive_subjects()`* walks backward. A good example uses a property `ex:parent`, the semantics of which are biological parentage. The *`transitive_objects()`* method would get all the ancestors of a particular person (all nodes such that there is a parent path between the person and the object). The *`transitive_subjects()`* method would get all the descendants of a particular person (all nodes such that there is a parent path between the node and the person). So, say that your URI is `ex:person`.

This example would get all of your (known) ancestors, and then get all the (known) descendants of your maternal grandmother.

> **Warning:** The `transitive_objects()` method has the start node as the *first* argument, but the `transitive_subjects()` method has the start node as the *second* argument.

User-defined transitive closures

The method *`transitiveClosure()`* returns transtive closures of user-defined functions.

CHAPTER

TWO

IN DEPTH

If you are familiar with RDF and are looking for details on how RDFLib handles RDF, these are for you.

2.1 RDF terms in rdflib

Terms are the kinds of objects that can appear in a quoted/asserted triples. Those that are part of core RDF concepts are: `Blank Node`, `URI Reference` and `Literal`, the latter consisting of a literal value and either a datatype or an **RFC 3066** language tag.

All terms in RDFLib are sub-classes of the *`rdflib.term.Identifier`* class.

Nodes are a subset of the Terms that the underlying store actually persists. The set of such Terms depends on whether or not the store is formula-aware. Stores that aren't formula-aware would only persist those terms core to the RDF Model, and those that are formula-aware would be able to persist the N3 extensions as well. However, utility terms that only serve the purpose for matching nodes by term-patterns probably will only be terms and not nodes.

2.1.1 BNodes

In RDF, a blank node (also called BNode) is a node in an RDF graph representing a resource for which a URI or literal is not given. The resource represented by a blank node is also called an anonymous resource. By RDF standard a blank node can only be used as subject or object in an RDF triple, although in some syntaxes like Notation 3 [1] it is acceptable to use a blank node as a predicate. If a blank node has a node ID (not all blank nodes are labelled in all RDF serializations), it is limited in scope to a serialization of a particular RDF graph, i.e. the node p1 in the subsequent example does not represent the same node as a node named p1 in any other graph –wikipedia

class rdflib.term.**BNode**

Blank Node: http://www.w3.org/TR/rdf-concepts/#section-blank-nodes

```
>>> from rdflib import BNode
>>> anode = BNode()
>>> anode
rdflib.term.BNode('AFwALAKU0')
>>> anode.n3()
u'_:AFwALAKU0'
```

2.1.2 URIRefs

A URI reference within an RDF graph is a Unicode string that does not contain any control characters (#x00 - #x1F, #x7F-#x9F) and would produce a valid URI character sequence representing an absolute URI with optional fragment identifier – W3 RDF Concepts

class rdflib.term.**URIRef**
RDF URI Reference: http://www.w3.org/TR/rdf-concepts/#section-Graph-URIref

```
>>> from rdflib import URIRef
>>> aref = URIRef()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() takes at least 2 arguments (1 given)
>>> aref = URIRef('')
>>> aref
rdflib.term.URIRef(u'')
>>> aref = URIRef('http://example.com')
>>> aref
rdflib.term.URIRef(u'http://example.com')
>>> aref.n3()
u'<http://example.com>'
```

2.1.3 Literals

Literals are the attribute values in RDF, for instance, a person's name, the date of birth, height, etc. Literals can have a data-type (i.e. this is a double) or a language tag (this label is in English).

class rdflib.term.**Literal**
RDF Literal: http://www.w3.org/TR/rdf-concepts/#section-Graph-Literal

The lexical value of the literal is the unicode object The interpreted, datatyped value is available from .value

Language tags must be valid according to :rfc:5646

For valid XSD datatypes, the lexical form is optionally normalized at construction time. Default behaviour is set by rdflib.NORMALIZE_LITERALS and can be overridden by the normalize parameter to __new__

Equality and hashing of Literals are done based on the lexical form, i.e.:

```
>>> from rdflib.namespace import XSD
```

```
>>> Literal('01')!=Literal('1') # clear - strings differ
True
```

but with data-type they get normalized:

```
>>> Literal('01', datatype=XSD.integer)!=Literal('1', datatype=XSD.integer)
False
```

unless disabled:

```
>>> Literal('01', datatype=XSD.integer, normalize=False)!=Literal('1',␣
↪datatype=XSD.integer)
True
```

Value based comparison is possible:

```
>>> Literal('01', datatype=XSD.integer).eq(Literal('1', datatype=XSD.float))
True
```

The eq method also provides limited support for basic python types:

```
>>> Literal(1).eq(1) # fine - int compatible with xsd:integer
True
>>> Literal('a').eq('b') # fine - str compatible with plain-lit
False
>>> Literal('a', datatype=XSD.string).eq('a') # fine - str compatible with␣
↪xsd:string
True
>>> Literal('a').eq(1) # not fine, int incompatible with plain-lit
NotImplemented
```

Greater-than/less-than ordering comparisons are also done in value space, when compatible datatypes are used. Incompatible datatypes are ordered by DT, or by lang-tag. For other nodes the ordering is None < BNode < URIRef < Literal

Any comparison with non-rdflib Node are "NotImplemented" In PY3 this is an error.

```
>>> from rdflib import Literal, XSD
>>> lit2006 = Literal('2006-01-01',datatype=XSD.date)
>>> lit2006.toPython()
datetime.date(2006, 1, 1)
>>> lit2006 < Literal('2007-01-01',datatype=XSD.date)
True
>>> Literal(datetime.utcnow()).datatype
rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#dateTime')
>>> Literal(1) > Literal(2) # by value
False
>>> Literal(1) > Literal(2.0) # by value
False
>>> Literal('1') > Literal(1) # by DT
True
>>> Literal('1') < Literal('1') # by lexical form
False
>>> Literal('a', lang='en') > Literal('a', lang='fr') # by lang-tag
False
>>> Literal(1) > URIRef('foo') # by node-type
True
```

The > < operators will eat this NotImplemented and throw a TypeError (py3k):

```
>>> Literal(1).__gt__(2.0)
NotImplemented
```

A literal in an RDF graph contains one or two named components.

All literals have a lexical form being a Unicode string, which SHOULD be in Normal Form C.

Plain literals have a lexical form and optionally a language tag as defined by **RFC 3066**, normalized to lowercase. An exception will be raised if illegal language-tags are passed to `rdflib.term.Literal.__init__()`.

Typed literals have a lexical form and a datatype URI being an RDF URI reference.

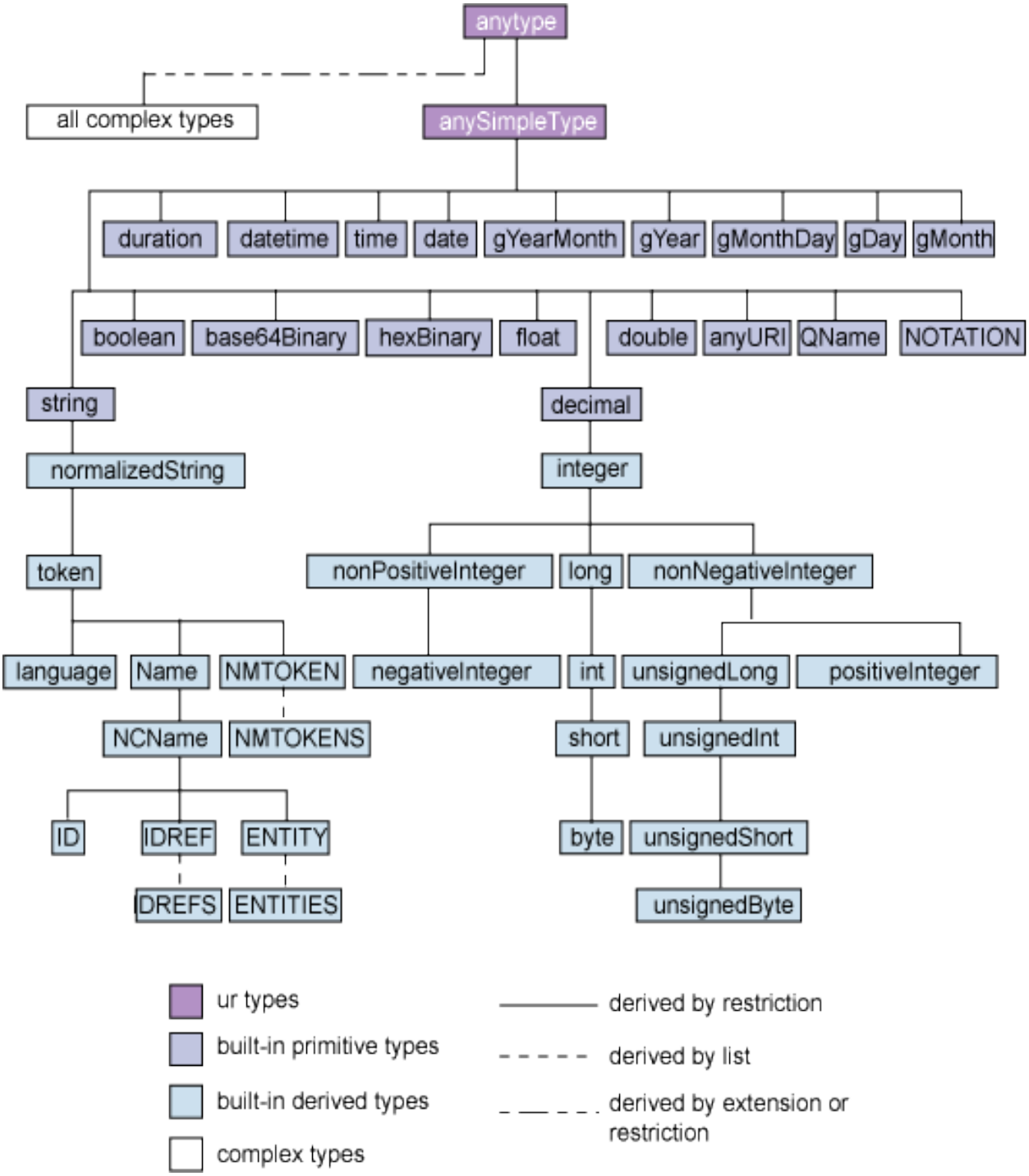
Note: When using the language tag, care must be taken not to confuse language with locale. The language tag relates

only to human language text. Presentational issues should be addressed in end-user applications.

Note: The case normalization of language tags is part of the description of the abstract syntax, and consequently the abstract behaviour of RDF applications. It does not constrain an RDF implementation to actually normalize the case. Crucially, the result of comparing two language tags should not be sensitive to the case of the original input. – RDF Concepts and Abstract Syntax

Python support

RDFLib Literals essentially behave like unicode characters with an XML Schema datatype or language attribute.



The class provides a mechanism to both convert Python literals (and their built-ins such as time/date/datetime) into equivalent RDF Literals and (conversely) convert Literals to their Python equivalent. This mapping to and from Python literals is done as follows:

XML Datatype	Python type
None	None[1]
xsd:time	time[2]
xsd:date	date
xsd:dateTime	datetime
xsd:string	None
xsd:normalizedString	None
xsd:token	None
xsd:language	None
xsd:boolean	boolean
xsd:decimal	Decimal
xsd:integer	long
xsd:nonPositiveInteger	int
xsd:long	long
xsd:nonNegativeInteger	int
xsd:negativeInteger	int
xsd:int	long
xsd:unsignedLong	long
xsd:positiveInteger	int
xsd:short	int
xsd:unsignedInt	long
xsd:byte	int
xsd:unsignedShort	int
xsd:unsignedByte	int
xsd:float	float
xsd:double	float
xsd:base64Binary	`base64`
xsd:anyURI	None
rdf:XMLLiteral	`xml.dom.minidom.Document`[3]
rdf:HTML	`xml.dom.minidom.DocumentFragment`

An appropriate data-type and lexical representation can be found using:

`rdflib.term.`**`_castPythonToLiteral`**(*obj*, *datatype*)
Casts a tuple of a python type and a special datatype URI to a tuple of the lexical value and a datatype URI (or None)

and the other direction with

`rdflib.term.`**`_castLexicalToPython`**(*lexical*, *datatype*)
Map a lexical form to the value-space for the given datatype :returns: a python object for the value or `None`

All this happens automatically when creating `Literal` objects by passing Python objects to the constructor, and you never have to do this manually.

You can add custom data-types with *`rdflib.term.bind()`*, see also *`examples.custom_datatype`*

[1] plain literals map directly to value space
[2] Date, time and datetime literals are mapped to Python instances using the isodate package).
[3] this is a bit dirty - by accident the `html5lib` parser produces `DocumentFragments`, and the xml parser `Documents`, letting us use this to decide what datatype when round-tripping.

2.2 Namespaces and Bindings

RDFLib provides several short-cuts to working with many URIs in the same namespace.

The *rdflib.namespace* defines the *rdflib.namespace.Namespace* class which lets you easily create URIs in a namespace:

```
from rdflib import Namespace

n = Namespace("http://example.org/")
n.Person # as attribute
# = rdflib.term.URIRef(u'http://example.org/Person')

n['first%20name'] # as item - for things that are not valid python identifiers
# = rdflib.term.URIRef(u'http://example.org/first%20name')
```

The `namespace` module also defines many common namespaces such as RDF, RDFS, OWL, FOAF, SKOS, etc.

Namespaces can also be associated with prefixes, in a *rdflib.namespace.NamespaceManager*, i.e. using `foaf` for `http://xmlns.com/foaf/0.1/`. Each RDFLib graph has a *namespace_manager* that keeps a list of namespace to prefix mappings. The namespace manager is populated when reading in RDF, and these prefixes are used when serialising RDF, or when parsing SPARQL queries. Additional prefixes can be bound with the `rdflib.graph.bind()` method.

2.2.1 NamespaceManager

Each graph comes with a NamespaceManager instance in the *namespace_manager* field; you can use the *bind* method of this instance to bind a prefix to a namespace URI:

```
myGraph.namespace_manager.bind('prefix', URIRef('scheme:my-namespace-uri:'))
myGraph.namespace_manager.bind('owl', OWL_NS, override=False)
```

It has a method to normalize a given url :

myGraph.namespace_manager.normalizeUri(t)

For simple output, or simple serialisation, you often want a nice readable representation of a term. All terms have a `.n3(namespace_manager = None)` method, which will return a suitable N3 format:

```
>>> from rdflib import Graph, URIRef, Literal, BNode
>>> from rdflib.namespace import FOAF, NamespaceManager

>>> person = URIRef('http://xmlns.com/foaf/0.1/Person')
>>> person.n3()
u'<http://xmlns.com/foaf/0.1/Person>'

>>> g = Graph()
>>> g.bind("foaf", FOAF)

>>> person.n3(g.namespace_manager)
u'foaf:Person'

>>> l = Literal(2)
>>> l.n3()
u'"2"^^<http://www.w3.org/2001/XMLSchema#integer>'
```

(continues on next page)

(continued from previous page)

```
>>> l.n3(g.namespace_manager)
u'"2"^^xsd:integer'
```

The namespace manage also has a useful method compute_qname g.namespace_manager.compute_qname(x) which takes an url and decomposes it into the parts:

self.assertEqual(g.compute_qname(URIRef("http://foo/bar#baz")), ("ns2", URIRef("http://foo/bar#"), "baz"))

2.2.2 Namespaces in SPARQL Queries

The `initNs` argument supplied to *`query()`* is a dictionary of namespaces to be expanded in the query string. If you pass no `initNs` argument, the namespaces registered with the graphs namespace_manager are used:

```
...
from rdflib.namespace import FOAF
graph.query('SELECT * WHERE { ?p a foaf:Person }', initNs={ 'foaf': FOAF })
```

In order to use an empty prefix (e.g. `?a :knows ?b`), use a `PREFIX` directive with no prefix in the SPARQL query to set a default namespace:

```
PREFIX : <http://xmlns.com/foaf/0.1/>
```

2.3 Persistence

RDFLib provides an *`abstracted Store API`* for persistence of RDF and Notation 3. The *`Graph`* class works with instances of this API (as the first argument to its constructor) for triple-based management of an RDF store including: garbage collection, transaction management, update, pattern matching, removal, length, and database management (*`open()`* / *`close()`* / *`destroy()`*).

Additional persistence mechanisms can be supported by implementing this API for a different store.

2.3.1 Stores currently shipped with core RDFLib

- *`Memory`* (not persistent!)
- *`Sleepycat`* (on disk persistence via Python's bsddb or bsddb3 packages)
- *`SPARQLStore`* - a read-only wrapper around a remote SPARQL Query endpoint.
- *`SPARQLUpdateStore`* - a read-write wrapper around a remote SPARQL query/update endpoint pair.

2.3.2 Usage

Most cases passing the name of the store to the Graph constructor is enough:

```
from rdflib import Graph

graph = Graph(store='Sleepycat')
```

Most store offering on-disk persistence will need to be opened before reading or writing. When peristing a triplestore (instead of a ConjuntiveGraph quadstore), you need to specify an identifier with which you can open the graph:

```
graph = Graph('Sleepycat', identifier='mygraph')

# first time create the store:
graph.open('/home/user/data/myRDFLibStore', create = True)

# work with the graph:
graph.add( mytriples )

# when done!
graph.close()
```

When done, `close()` must be called to free the resources associated with the store.

2.3.3 Additional store plugins

More store implementations are available in RDFLib extension projects:

- rdflib-sqlalchemy, which supports stored on a wide-variety of RDBMs backends,
- rdflib-leveldb - a store on to of Google's LevelDB key-value store.
- rdflib-kyotocabinet - a store on to of the Kyoto Cabinet key-value store.

2.3.4 Example

- `examples.sleepycat_example` contains an example for using a Sleepycat store.
- `examples.sparqlstore_example` contains an example for using a SPARQLStore.

2.4 Merging graphs

A merge of a set of RDF graphs is defined as follows. If the graphs in the set have no blank nodes in common, then the union of the graphs is a merge; if they do share blank nodes, then it is the union of a set of graphs that is obtained by replacing the graphs in the set by equivalent graphs that share no blank nodes. This is often described by saying that the blank nodes have been 'standardized apart'. It is easy to see that any two merges are equivalent, so we will refer to the merge, following the convention on equivalent graphs. Using the convention on equivalent graphs and identity, any graph in the original set is considered to be a subgraph of the merge.

One does not, in general, obtain the merge of a set of graphs by concatenating their corresponding N-Triples documents and constructing the graph described by the merged document. If some of the documents use the same node identifiers, the merged document will describe a graph in which some of the blank nodes have been 'accidentally' identified. To merge N-Triples documents it is necessary to check

> if the same nodeID is used in two or more documents, and to replace it with a distinct nodeID in each of them, before merging the documents. Similar cautions apply to merging graphs described by RDF/XML documents which contain nodeIDs

(copied directly from http://www.w3.org/TR/rdf-mt/#graphdefs)

In RDFLib, blank nodes are given unique IDs when parsing, so graph merging can be done by simply reading several files into the same graph:

```
from rdflib import Graph

graph = Graph()

graph.parse(input1)
graph.parse(input2)
```

`graph` now contains the merged graph of `input1` and `input2`.

Note: However, the set-theoretic graph operations in RDFLib are assumed to be performed in sub-graphs of some larger data-base (for instance, in the context of a *`ConjunctiveGraph`*) and assume shared blank node IDs, and therefore do NOT do *correct* merging, i.e.:

```
from rdflib import Graph

g1 = Graph()
g1.parse(input1)

g2 = Graph()
g2.parse(input2)

graph = g1 + g2
```

May cause unwanted collisions of blank-nodes in `graph`.

2.5 Upgrading 4.2.2 to 5.0.0

RDFLib version 5.0.0 appeared over 3 years after the previous release, 4.2.2 and contains a large number of both enhancements and bug fixes. Fundamentally though, 5.0.0 is compatible with 4.2.2.

2.5.1 Major Changes

Literal Ordering

Literal total ordering PR #793 is implemented. That means all literals can now be compared to be greater than or less than any other literal. This is required for implementing some specific SPARQL features, but it is counter-intuitive to those who are expecting a TypeError when certain normally-incompatible types are compared. For example, comparing a `Literal(int(1), datatype=xsd:integer)` to `Literal(datetime.date(10, 01,2020), datatype=xsd:date)` using a > or < operator in rdflib 4.2.2 and earlier, would normally throw a TypeError, however in rdflib 5.0.0 this operation now returns a True or False according to the Literal Total Ordering according the rules outlined in PR #793

Removed RDF Parsers

The RDFa and Microdata format RDF parsers were removed from rdflib. There are still other python libraries available to implement these parsers.

2.5.2 All Changes

This list has been assembled from Pull Request and commit information.

General Bugs Fixed:

- Pr 451 redux PR #978
- NTriples fails to parse URIs with only a scheme ISSUE #920 PR #974
- cannot clone it on windows - Remove colons from test result files. Fix #901. ISSUE #901 PR #971
- Add requirement for requests to setup.py PR #969
- fixed URIRef including native unicode characters PR #961
- DCTERMS.format not working ISSUE #932
- infixowl.manchesterSyntax do not encode strings PR #906
- Fix blank node label to not contain '_:' during parsing PR #886
- rename new SPARQLWrapper to SPARQLConnector PR #872
- Fix #859. Unquote and Uriquote Literal Datatype. PR #860
- Parsing nquads ISSUE #786
- ntriples spec allows for upper-cased lang tag, fixes #782 PR #784
- Error parsing N-Triple file using RDFlib ISSUE #782
- Adds escaped single quote to literal parser PR #736
- N3 parse error on single quote within single quotes ISSUE #732
- Fixed #725 PR #730
- test for issue #725: canonicalization collapses BNodes PR #726
- RGDA1 graph canonicalization sometimes still collapses distinct BNodes ISSUE #725
- Accept header should use a q parameter PR #720
- Added test for Issue #682 and fixed. PR #718
- Incompatibility with Python3: unichr ISSUE #687
- namespace.py include colon in ALLOWED_NAME_CHARS PR #663
- namespace.py fix compute_qname missing namespaces PR #649
- RDFa parsing Error! *__init__()* got an unexpected keyword argument 'encoding' ISSUE #639
- Bugfix: *term.Literal.__add__* PR #451
- fixup of #443 PR #445
- Microdata to rdf second edition bak PR #444

Enhanced Features:

- Register additional serializer plugins for SPARQL mime types. PR #987
- Pr 388 redux PR #979
- Allows RDF terms introduced by JSON-LD 1.1 PR #970
- make SPARQLConnector work with DBpedia PR #941
- ClosedNamespace returns right exception for way of access PR #866
- Not adding all namespaces for n3 serializer PR #832
- Adds basic support of xsd:duration PR #808
- Add possibility to set authority and basepath to skolemize graph PR #807
- Change notation3 list realization to non-recursive function. PR #805
- Suppress warning for not using custom encoding. PR #800
- Add support to parsing large xml inputs ISSUE #749 PR #750
- improve hash efficiency by directly using str/unicode hash PR #746
- Added the csvw prefix to the RDFa initial context. PR #594
- syncing changes from pyMicrodata PR #587
- Microdata parser: updated the parser to the latest version of the microdata->rdf note (published in December 2014) PR #443
- Literal.toPython() support for xsd:hexBinary PR #388

SPARQL Fixes:

- Total order patch patch PR #862
- use <<= instead of deprecated << PR #861
- Fix #847 PR #856
- RDF Literal "1"^^xsd:boolean should _not_ coerce to True ISSUE #847
- Makes NOW() return an UTC date PR #844
- NOW() SPARQL should return an xsd:dateTime with a timezone ISSUE #843
- fix property paths bug: issue #715 PR #822 ISSUE #715
- MulPath: correct behaviour of n3() PR #820
- Literal total ordering PR #793
- Remove SPARQLWrapper dependency PR #744
- made UNION faster by not preventing duplicates PR #741
- added a hook to add custom functions to SPARQL PR #723
- Issue714 PR #717
- Use <<= instead of deprecated << in SPARQL parser PR #417
- Custom FILTER function for SPARQL engine ISSUE #274

Code Quality and Cleanups:

- a slightly opinionated autopep8 run PR #870
- remove rdfa and microdata parsers from core RDFLib PR #828
- ClosedNamespace KeyError -> AttributeError PR #827
- typo in rdflib/plugins/sparql/update.py ISSUE #760
- Fix logging in interactive mode PR #731
- make namespace module flake8-compliant, change exceptions in that mod... PR #711
- delete ez_setup.py? ISSUE #669
- code duplication issue between rdflib and pymicrodata ISSUE #582
- Transition from 2to3 to use of six.py to be merged in 5.0.0-dev PR #519
- sparqlstore drop deprecated methods and args PR #516
- python3 code seems shockingly inefficient ISSUE #440
- removed md5_term_hash, fixes #240 PR #439 ISSUE #240

Testing:

- 3.7 for travis PR #864
- Added trig unit tests to highlight some current parsing/serializing issues PR #431

Documentation Fixes:

- Fix a doc string in the query module PR #976
- setup.py: Make the license field use an SPDX identifier PR #789
- Update README.md PR #764
- Update namespaces_and_bindings.rst PR #757
- DOC: README.md: rdflib-jsonld, https uris PR #712
- make doctest support py2/py3 ISSUE #707
- *pip install rdflib* (as per README.md) gets OSError on Mint 18.1 ISSUE #704 PR #717
- Use <<= instead of deprecated << in SPARQL parser PR #417
- Custom FILTER function for SPARQL engine ISSUE #274

Code Quality and Cleanups:

- a slightly opinionated autopep8 run PR #870
- remove rdfa and microdata parsers from core RDFLib PR #828
- ClosedNamespace KeyError -> AttributeError PR #827
- typo in rdflib/plugins/sparql/update.py ISSUE #760
- Fix logging in interactive mode PR #731
- make namespace module flake8-compliant, change exceptions in that mod... PR #711
- delete ez_setup.py? ISSUE #669
- code duplication issue between rdflib and pymicrodata ISSUE #582
- Transition from 2to3 to use of six.py to be merged in 5.0.0-dev PR #519
- sparqlstore drop deprecated methods and args PR #516
- python3 code seems shockingly inefficient ISSUE #440
- removed md5_term_hash, fixes #240 PR #439 ISSUE #240

Testing:

- 3.7 for travis PR #864
- Added trig unit tests to highlight some current parsing/serializing issues PR #431

Documentation Fixes:

- Fix a doc string in the query module PR #976
- setup.py: Make the license field use an SPDX identifier PR #789
- Update README.md PR #764
- Update namespaces_and_bindings.rst PR #757
- DOC: README.md: rdflib-jsonld, https uris PR #712
- make doctest support py2/py3 ISSUE #707
- *pip install rdflib* (as per README.md) gets OSError on Mint 18.1 ISSUE #704

CHAPTER

THREE

REFERENCE

The nitty-gritty details of everything.

API reference:

3.1 rdflib

3.1.1 rdflib package

Subpackages

rdflib.extras package

Submodules

rdflib.extras.cmdlineutils module

`rdflib.extras.cmdlineutils.`**`main`**(*target*, *_help=<function _help>*, *options=''*, *stdin=True*)
A main function for tools that read RDF from files given on commandline or from STDIN (if stdin parameter is true)

rdflib.extras.describer module

A Describer is a stateful utility for creating RDF statements in a semi-declarative manner. It has methods for creating literal values, rel and rev resource relations (somewhat resembling RDFa).

The *rel* and `rev` methods return a context manager which sets the current about to the referenced resource for the context scope (for use with the `with` statement).

Full example in the `to_rdf` method below:

```
>>> import datetime
>>> from rdflib.graph import Graph
>>> from rdflib.namespace import Namespace, RDFS, FOAF
>>>
>>> ORG_URI = "http://example.org/"
>>>
>>> CV = Namespace("http://purl.org/captsolo/resume-rdf/0.2/cv#")
>>>
```

(continues on next page)

(continued from previous page)

```
>>> class Person(object):
...     def __init__(self):
...         self.first_name = u"Some"
...         self.last_name = u"Body"
...         self.username = "some1"
...         self.presentation = u"Just a Python & RDF hacker."
...         self.image = "/images/persons/" + self.username + ".jpg"
...         self.site = "http://example.net/"
...         self.start_date = datetime.date(2009, 9, 4)
...     def get_full_name(self):
...         return u" ".join([self.first_name, self.last_name])
...     def get_absolute_url(self):
...         return "/persons/" + self.username
...     def get_thumbnail_url(self):
...         return self.image.replace('.jpg', '-thumb.jpg')
...
...     def to_rdf(self):
...         graph = Graph()
...         graph.bind('foaf', FOAF)
...         graph.bind('cv', CV)
...         lang = 'en'
...         d = Describer(graph, base=ORG_URI)
...         d.about(self.get_absolute_url()+'#person')
...         d.rdftype(FOAF.Person)
...         d.value(FOAF.name, self.get_full_name())
...         d.value(FOAF.givenName, self.first_name)
...         d.value(FOAF.familyName, self.last_name)
...         d.rel(FOAF.homepage, self.site)
...         d.value(RDFS.comment, self.presentation, lang=lang)
...         with d.rel(FOAF.depiction, self.image):
...             d.rdftype(FOAF.Image)
...             d.rel(FOAF.thumbnail, self.get_thumbnail_url())
...         with d.rev(CV.aboutPerson):
...             d.rdftype(CV.CV)
...             with d.rel(CV.hasWorkHistory):
...                 d.value(CV.startDate, self.start_date)
...                 d.rel(CV.employedIn, ORG_URI+"#company")
...         return graph
...
>>> person_graph = Person().to_rdf()
>>> expected = Graph().parse(data='''<?xml version="1.0" encoding="utf-8"?>
... <rdf:RDF
...   xmlns:foaf="http://xmlns.com/foaf/0.1/"
...   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
...   xmlns:cv="http://purl.org/captsolo/resume-rdf/0.2/cv#"
...   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
...   <foaf:Person rdf:about="http://example.org/persons/some1#person">
...     <foaf:name>Some Body</foaf:name>
...     <foaf:givenName>Some</foaf:givenName>
...     <foaf:familyName>Body</foaf:familyName>
...     <foaf:depiction>
...       <foaf:Image
...         rdf:about=
...             "http://example.org/images/persons/some1.jpg">
...         <foaf:thumbnail
...         rdf:resource=
...             "http://example.org/images/persons/some1-thumb.jpg"/>
```

(continues on next page)

(continued from previous page)

```
...       </foaf:Image>
...     </foaf:depiction>
...     <rdfs:comment xml:lang="en">
...             Just a Python &amp;amp; RDF hacker.
...     </rdfs:comment>
...     <foaf:homepage rdf:resource="http://example.net/"/>
...   </foaf:Person>
...   <cv:CV>
...     <cv:aboutPerson
...         rdf:resource="http://example.org/persons/some1#person">
...     </cv:aboutPerson>
...     <cv:hasWorkHistory>
...       <rdf:Description>
...         <cv:startDate
...             rdf:datatype="http://www.w3.org/2001/XMLSchema#date"
...             >2009-09-04</cv:startDate>
...         <cv:employedIn rdf:resource="http://example.org/#company"/>
...       </rdf:Description>
...     </cv:hasWorkHistory>
...   </cv:CV>
... </rdf:RDF>
... ''')
>>>
>>> from rdflib.compare import isomorphic
>>> isomorphic(person_graph, expected)
True
```

class rdflib.extras.describer.**Describer**(*graph=None*, *about=None*, *base=None*)

Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.extras.describer', '__init__': <funct

__init__(*graph=None*, *about=None*, *base=None*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.extras.describer'

__weakref__

list of weak references to the object (if defined)

about(*subject*, ***kws*)

Sets the current subject. Will convert the given object into an URIRef if it's not an Identifier.

Usage:

```
>>> d = Describer()
>>> d._current()
rdflib.term.BNode(...)
>>> d.about("http://example.org/")
>>> d._current()
rdflib.term.URIRef(u'http://example.org/')
```

rdftype(*t*)

Shorthand for setting rdf:type of the current subject.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
```

(continues on next page)

(continued from previous page)

```
>>> d = Describer(about="http://example.org/")
>>> d.rdftype(RDFS.Resource)
>>> (URIRef('http://example.org/'),
...     RDF.type, RDFS.Resource) in d.graph
True
```

rel (*p*, *o=None*, ***kws*)

Set an object for the given property. Will convert the given object into an `URIRef` if it's not an `Identifier`. If none is given, a new `BNode` is used.

Returns a context manager for use in a `with` block, within which the given object is used as current subject.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
>>> d = Describer(about="/", base="http://example.org/")
>>> _ctxt = d.rel(RDFS.seeAlso, "/about")
>>> d.graph.value(URIRef('http://example.org/'), RDFS.seeAlso)
rdflib.term.URIRef(u'http://example.org/about')

>>> with d.rel(RDFS.seeAlso, "/more"):
...     d.value(RDFS.label, "More")
>>> (URIRef('http://example.org/'), RDFS.seeAlso,
...         URIRef('http://example.org/more')) in d.graph
True
>>> d.graph.value(URIRef('http://example.org/more'), RDFS.label)
rdflib.term.Literal(u'More')
```

rev (*p*, *s=None*, ***kws*)

Same as `rel`, but uses current subject as *object* of the relation. The given resource is still used as subject in the returned context manager.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
>>> d = Describer(about="http://example.org/")
>>> with d.rev(RDFS.seeAlso, "http://example.net/"):
...     d.value(RDFS.label, "Net")
>>> (URIRef('http://example.net/'), RDFS.seeAlso,
...         URIRef('http://example.org/')) in d.graph
True
>>> d.graph.value(URIRef('http://example.net/'), RDFS.label)
rdflib.term.Literal(u'Net')
```

value (*p*, *v*, ***kws*)

Set a literal value for the given property. Will cast the value to an `Literal` if a plain literal is given.

Usage:

```
>>> from rdflib import URIRef
>>> from rdflib.namespace import RDF, RDFS
>>> d = Describer(about="http://example.org/")
>>> d.value(RDFS.label, "Example")
>>> d.graph.value(URIRef('http://example.org/'), RDFS.label)
rdflib.term.Literal(u'Example')
```

rdflib.extras.describer.**cast_identifier**(*ref*, ***kws*)

rdflib.extras.describer.**cast_value**(*v*, ***kws*)

rdflib.extras.external_graph_libs module

rdflib.extras.external_graph_libs.**rdflib_to_graphtool**(*graph*, *v_prop_names=['term']*, *e_prop_names=['term']*, *transform_s=<function <lambda>>*, *transform_p=<function <lambda>>*, *transform_o=<function <lambda>>*)

Converts the given graph into a graph_tool.Graph().

The subjects and objects are the later vertices of the Graph. The predicates become edges.

Arguments: graph: a rdflib.Graph. v_prop_names: a list of names for the vertex properties. The default is set to [‘term’] (see transform_s, transform_o below).

e_prop_names: a list of names for the edge properties. transform_s: callable with s, p, o input. Should return a dictionary containing a value for each name in v_prop_names. By default is set to {‘term’: s} which in combination with v_prop_names = [‘term’] adds s as ‘term’ property to the generated vertex for s.

transform_p: similar to transform_s, but wrt. e_prop_names. By default returns {‘term’: p} which adds p as a property to the generated edge between the vertex for s and the vertex for o.

transform_o: similar to transform_s.

Returns: graph_tool.Graph()

```
>>> from rdflib import Graph, URIRef, Literal
>>> g = Graph()
>>> a, b, l = URIRef('a'), URIRef('b'), Literal('l')
>>> p, q = URIRef('p'), URIRef('q')
>>> edges = [(a, p, b), (a, q, b), (b, p, a), (b, p, l)]
>>> for t in edges:
...     g.add(t)
...
>>> mdg = rdflib_to_graphtool(g)
>>> len(list(mdg.edges()))
4
>>> from graph_tool import util as gt_util
>>> vpterm = mdg.vertex_properties['term']
>>> va = gt_util.find_vertex(mdg, vpterm, a)[0]
>>> vb = gt_util.find_vertex(mdg, vpterm, b)[0]
>>> vl = gt_util.find_vertex(mdg, vpterm, l)[0]
>>> (va, vb) in [(e.source(), e.target()) for e in list(mdg.edges())]
True
>>> epterm = mdg.edge_properties['term']
>>> len(list(gt_util.find_edge(mdg, epterm, p))) == 3
True
```

(continues on next page)

(continued from previous page)

```
>>> len(list(gt_util.find_edge(mdg, epterm, q))) == 1
True
```

```
>>> mdg = rdflib_to_graphtool(
...     g,
...     e_prop_names=[str('name')],
...     transform_p=lambda s, p, o: {str('name'): unicode(p)})
>>> epterm = mdg.edge_properties['name']
>>> len(list(gt_util.find_edge(mdg, epterm, unicode(p)))) == 3
True
>>> len(list(gt_util.find_edge(mdg, epterm, unicode(q)))) == 1
True
```

rdflib.extras.external_graph_libs.**rdflib_to_networkx_digraph**(*graph*, *calc_weights=True*, *edge_attrs=<function <lambda>>*, ***kwds*)

Converts the given graph into a networkx.DiGraph.

As an rdflib.Graph() can contain multiple edges between nodes, by default adds the a 'triples' attribute to the single DiGraph edge with a list of all triples between s and o. Also by default calculates the edge weight as the length of triples.

Args: graph: a rdflib.Graph. calc_weights: If true calculate multi-graph edge-count as edge 'weight' edge_attrs: Callable to construct later edge_attributes. It receives

3 variables (s, p, o) and should construct a dictionary that is passed to networkx's add_edge(s, o, **attrs) function.

By default this will include setting the 'triples' attribute here, which is treated specially by us to be merged. Other attributes of multi-edges will only contain the attributes of the first edge. If you don't want the 'triples' attribute for tracking, set this to *lambda s, p, o: {}*.

Returns: networkx.DiGraph

```
>>> from rdflib import Graph, URIRef, Literal
>>> g = Graph()
>>> a, b, l = URIRef('a'), URIRef('b'), Literal('l')
>>> p, q = URIRef('p'), URIRef('q')
>>> edges = [(a, p, b), (a, q, b), (b, p, a), (b, p, l)]
>>> for t in edges:
...     g.add(t)
...
>>> dg = rdflib_to_networkx_digraph(g)
>>> dg[a][b]['weight']
2
>>> sorted(dg[a][b]['triples']) == [(a, p, b), (a, q, b)]
True
>>> len(dg.edges())
3
>>> dg.size()
3
>>> dg.size(weight='weight')
4.0
```

```
>>> dg = rdflib_to_networkx_graph(g, False, edge_attrs=lambda s,p,o:{})
>>> 'weight' in dg[a][b]
False
>>> 'triples' in dg[a][b]
False
```

rdflib.extras.external_graph_libs.**rdflib_to_networkx_graph**(*graph*, *calc_weights=True*, *edge_attrs=<function <lambda>>*, ***kwds*)

Converts the given graph into a networkx.Graph.

As an rdflib.Graph() can contain multiple directed edges between nodes, by default adds the a 'triples' attribute to the single DiGraph edge with a list of triples between s and o in graph. Also by default calculates the edge weight as the len(triples).

Args: graph: a rdflib.Graph. calc_weights: If true calculate multi-graph edge-count as edge 'weight' edge_attrs: Callable to construct later edge_attributes. It receives

3 variables (s, p, o) and should construct a dictionary that is passed to networkx's add_edge(s, o, **attrs) function.

By default this will include setting the 'triples' attribute here, which is treated specially by us to be merged. Other attributes of multi-edges will only contain the attributes of the first edge. If you don't want the 'triples' attribute for tracking, set this to *lambda s, p, o: {}*.

Returns: networkx.Graph

```
>>> from rdflib import Graph, URIRef, Literal
>>> g = Graph()
>>> a, b, l = URIRef('a'), URIRef('b'), Literal('l')
>>> p, q = URIRef('p'), URIRef('q')
>>> edges = [(a, p, b), (a, q, b), (b, p, a), (b, p, l)]
>>> for t in edges:
...     g.add(t)
...
>>> ug = rdflib_to_networkx_graph(g)
>>> ug[a][b]['weight']
3
>>> sorted(ug[a][b]['triples']) == [(a, p, b), (a, q, b), (b, p, a)]
True
>>> len(ug.edges())
2
>>> ug.size()
2
>>> ug.size(weight='weight')
4.0
```

```
>>> ug = rdflib_to_networkx_graph(g, False, edge_attrs=lambda s,p,o:{})
>>> 'weight' in ug[a][b]
False
>>> 'triples' in ug[a][b]
False
```

rdflib.extras.external_graph_libs.**rdflib_to_networkx_multidigraph**(*graph*, *edge_attrs=<function <lambda>>*, ***kwds*)

Converts the given graph into a networkx.MultiDiGraph.

The subjects and objects are the later nodes of the MultiDiGraph. The predicates are used as edge keys (to identify multi-edges).

Arguments: graph: a rdflib.Graph. edge_attrs: Callable to construct later edge_attributes. It receives

3 variables (s, p, o) and should construct a dictionary that is passed to networkx's add_edge(s, o, **attrs) function.

By default this will include setting the MultiDiGraph key=p here. If you don't want to be able to re-identify the edge later on, you can set this to *lambda s, p, o: {}*. In this case MultiDiGraph's default (increasing ints) will be used.

Returns: networkx.MultiDiGraph

```
>>> from rdflib import Graph, URIRef, Literal
>>> g = Graph()
>>> a, b, l = URIRef('a'), URIRef('b'), Literal('l')
>>> p, q = URIRef('p'), URIRef('q')
>>> edges = [(a, p, b), (a, q, b), (b, p, a), (b, p, l)]
>>> for t in edges:
...     g.add(t)
...
>>> mdg = rdflib_to_networkx_multidigraph(g)
>>> len(mdg.edges())
4
>>> mdg.has_edge(a, b)
True
>>> mdg.has_edge(a, b, key=p)
True
>>> mdg.has_edge(a, b, key=q)
True
```

```
>>> mdg = rdflib_to_networkx_multidigraph(g, edge_attrs=lambda s,p,o: {})
>>> mdg.has_edge(a, b, key=0)
True
>>> mdg.has_edge(a, b, key=1)
True
```

rdflib.extras.infixowl module

RDFLib Python binding for OWL Abstract Syntax

see: **http://www.w3.org/TR/owl-semantics/syntax.html** http://owl-workshop.man.ac.uk/acceptedLong/submission_9.pdf

3.2.3 Axioms for complete classes without using owl:equivalentClass

Named class description of type 2 (with owl:oneOf) or type 4-6 (with owl:intersectionOf, owl:unionOf or owl:complementOf

Uses Manchester Syntax for __repr__

```
>>> exNs = Namespace('http://example.com/')
>>> namespace_manager = NamespaceManager(Graph())
>>> namespace_manager.bind('ex', exNs, override=False)
>>> namespace_manager.bind('owl', OWL_NS, override=False)
```

(continues on next page)

(continued from previous page)

```
>>> g = Graph()
>>> g.namespace_manager = namespace_manager
```

Now we have an empty graph, we can construct OWL classes in it using the Python classes defined in this module

```
>>> a = Class(exNs.Opera, graph=g)
```

Now we can assert rdfs:subClassOf and owl:equivalentClass relationships (in the underlying graph) with other classes using the 'subClassOf' and 'equivalentClass' descriptors which can be set to a list of objects for the corresponding predicates.

```
>>> a.subClassOf = [exNs.MusicalWork]
```

We can then access the rdfs:subClassOf relationships

```
>>> print(list(a.subClassOf))
[Class: ex:MusicalWork ]
```

This can also be used against already populated graphs:

```
>>> owlGraph = Graph().parse(OWL_NS)
>>> namespace_manager.bind('owl', OWL_NS, override=False)
>>> owlGraph.namespace_manager = namespace_manager
>>> list(Class(OWL_NS.Class, graph=owlGraph).subClassOf)
[Class: rdfs:Class ]
```

Operators are also available. For instance we can add ex:Opera to the extension of the ex:CreativeWork class via the '+=' operator

```
>>> a
Class: ex:Opera SubClassOf: ex:MusicalWork
>>> b = Class(exNs.CreativeWork, graph=g)
>>> b += a
>>> print(sorted(a.subClassOf, key=lambda c:c.identifier))
[Class: ex:CreativeWork , Class: ex:MusicalWork ]
```

And we can then remove it from the extension as well

```
>>> b -= a
>>> a
Class: ex:Opera SubClassOf: ex:MusicalWork
```

Boolean class constructions can also be created with Python operators. For example, The | operator can be used to construct a class consisting of a owl:unionOf the operands:

```
>>> c =  a | b | Class(exNs.Work, graph=g)
>>> c
( ex:Opera OR ex:CreativeWork OR ex:Work )
```

Boolean class expressions can also be operated as lists (using python list operators)

```
>>> del c[c.index(Class(exNs.Work, graph=g))]
>>> c
( ex:Opera OR ex:CreativeWork )
```

The '&' operator can be used to construct class intersection:

```
>>> woman = Class(exNs.Female, graph=g) & Class(exNs.Human, graph=g)
>>> woman.identifier = exNs.Woman
>>> woman
( ex:Female AND ex:Human )
>>> len(woman)
2
```

Enumerated classes can also be manipulated

```
>>> contList = [Class(exNs.Africa, graph=g), Class(exNs.NorthAmerica, graph=g)]
>>> EnumeratedClass(members=contList, graph=g)
{ ex:Africa ex:NorthAmerica }
```

owl:Restrictions can also be instantiated:

```
>>> Restriction(exNs.hasParent, graph=g, allValuesFrom=exNs.Human)
( ex:hasParent ONLY ex:Human )
```

Restrictions can also be created using Manchester OWL syntax in 'colloquial' Python >>> exNs.hasParent | some | Class(exNs.Physician, graph=g) #doctest: +SKIP (ex:hasParent SOME ex:Physician)

```
>>> Property(exNs.hasParent,graph=g) | max | Literal(1)
( ex:hasParent MAX 1 )
```

```
>>> print(g.serialize(format='pretty-xml'))
```

rdflib.extras.infixowl.**AllClasses**(*graph*)

rdflib.extras.infixowl.**AllDifferent**(*members*)
DisjointClasses(' description description { description } ')'

rdflib.extras.infixowl.**AllProperties**(*graph*)

class rdflib.extras.infixowl.**AnnotatableTerms**(*identifier*, *graph=None*, *nameAnnotation=None*, *nameIsLabel=False*)

Bases: *rdflib.extras.infixowl.Individual*

Terms in an OWL ontology with rdfs:label and rdfs:comment

__init__(*identifier*, *graph=None*, *nameAnnotation=None*, *nameIsLabel=False*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.extras.infixowl'

property comment

handleAnnotation(*val*)

property label

property seeAlso

setupACEAnnotations()

class rdflib.extras.infixowl.**BooleanClass**(*identifier=None*, *operator=rdflib.term.URIRef('http://www.w3.org/2002/07/owl#intersectionOf')*, *members=None*, *graph=None*)

Bases: *rdflib.extras.infixowl.OWLRDFListProxy*, *rdflib.extras.infixowl.Class*

See: http://www.w3.org/TR/owl-ref/#Boolean

owl:complementOf is an attribute of Class, however

__init__(*identifier=None*, *operator=rdflib.term.URIRef('http://www.w3.org/2002/07/owl#intersectionOf')*, *members=None*, *graph=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.extras.infixowl'

__or__(*other*)
Adds other to the list and returns self

__repr__()
Returns the Manchester Syntax equivalent for this class

changeOperator(*newOperator*)
Converts a unionOf / intersectionOf class expression into one that instead uses the given operator

```
>>> testGraph = Graph()
>>> Individual.factoryGraph = testGraph
>>> EX = Namespace("http://example.com/")
>>> namespace_manager = NamespaceManager(Graph())
>>> namespace_manager.bind('ex', EX, override=False)
>>> testGraph.namespace_manager = namespace_manager
>>> fire = Class(EX.Fire)
>>> water = Class(EX.Water)
>>> testClass = BooleanClass(members=[fire,water])
>>> testClass
( ex:Fire AND ex:Water )
>>> testClass.changeOperator(OWL_NS.unionOf)
>>> testClass
( ex:Fire OR ex:Water )
>>> try: testClass.changeOperator(OWL_NS.unionOf)
... except Exception as e: print(e)
The new operator is already being used!
```

copy()
Create a copy of this class

getIntersections = <rdflib.extras.infixowl.Callable object>

getUnions = <rdflib.extras.infixowl.Callable object>

isPrimitive()

serialize(*graph*)

class rdflib.extras.infixowl.**Callable**(*anycallable*)
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.extras.infixowl', '__init__': <functi

__init__(*anycallable*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.extras.infixowl'

__weakref__
list of weak references to the object (if defined)

rdflib.extras.infixowl.**CastClass**(*c*, *graph=None*)

class rdflib.extras.infixowl.**Class**(*identifier=None*, *subClassOf=None*, *equivalentClass=None*, *disjointWith=None*, *complementOf=None*, *graph=None*, *skipOWLClassMembership=False*, *comment=None*, *nounAnnotations=None*, *nameAnnotation=None*, *nameIsLabel=False*)

Bases: *rdflib.extras.infixowl.AnnotatableTerms*

'General form' for classes:

The Manchester Syntax (supported in Protege) is used as the basis for the form of this class

See: http://owl-workshop.man.ac.uk/acceptedLong/submission_9.pdf:

[Annotation] 'Class:' classID {Annotation

(('SubClassOf:' ClassExpression) | ('EquivalentTo' ClassExpression) | ('DisjointWith' ClassExpression)) }

Appropriate excerpts from OWL Reference:

".. Subclass axioms provide us with partial definitions: they represent necessary but not sufficient conditions for establishing class membership of an individual."

".. A class axiom may contain (multiple) owl:equivalentClass statements"

"..A class axiom may also contain (multiple) owl:disjointWith statements.."

"..An owl:complementOf property links a class to precisely one class description."

__and__(*other*)

Construct an anonymous class description consisting of the intersection of this class and 'other' and return it

```
>>> exNs = Namespace('http://example.com/')
>>> namespace_manager = NamespaceManager(Graph())
>>> namespace_manager.bind('ex', exNs, override=False)
>>> namespace_manager.bind('owl', OWL_NS, override=False)
>>> g = Graph()
>>> g.namespace_manager = namespace_manager
```

Chaining 3 intersections

```
>>> female = Class(exNs.Female, graph=g)
>>> human = Class(exNs.Human, graph=g)
>>> youngPerson = Class(exNs.YoungPerson, graph=g)
>>> youngWoman = female & human & youngPerson
>>> youngWoman
ex:YoungPerson THAT ( ex:Female AND ex:Human )
>>> isinstance(youngWoman, BooleanClass)
True
>>> isinstance(youngWoman.identifier, BNode)
True
```

__eq__(*other*)

Return self==value.

__hash__()

```
>>> b=Class(OWL_NS.Restriction)
>>> c=Class(OWL_NS.Restriction)
>>> len(set([b,c]))
1
```

__iadd__(*other*)

__init__(*identifier=None*, *subClassOf=None*, *equivalentClass=None*, *disjointWith=None*, *complementOf=None*, *graph=None*, *skipOWLClassMembership=False*, *comment=None*, *nounAnnotations=None*, *nameAnnotation=None*, *nameIsLabel=False*)
Initialize self. See help(type(self)) for accurate signature.

__invert__()
Shorthand for Manchester syntax's not operator

__isub__(*other*)

__module__ = 'rdflib.extras.infixowl'

__or__(*other*)
Construct an anonymous class description consisting of the union of this class and 'other' and return it

__repr__(*full=False*, *normalization=True*)
Returns the Manchester Syntax equivalent for this class

property annotation

property complementOf

property disjointWith

property equivalentClass

property extent

property extentQuery

isPrimitive()

property parents
computed attributes that returns a generator over taxonomic 'parents' by disjunction, conjunction, and subsumption

```
>>> from rdflib.util import first
>>> exNs = Namespace('http://example.com/')
>>> namespace_manager = NamespaceManager(Graph())
>>> namespace_manager.bind('ex', exNs, override=False)
>>> namespace_manager.bind('owl', OWL_NS, override=False)
>>> g = Graph()
>>> g.namespace_manager = namespace_manager
>>> Individual.factoryGraph = g
>>> brother = Class(exNs.Brother)
>>> sister = Class(exNs.Sister)
>>> sibling = brother | sister
>>> sibling.identifier = exNs.Sibling
>>> sibling
( ex:Brother OR ex:Sister )
>>> first(brother.parents)
Class: ex:Sibling EquivalentTo: ( ex:Brother OR ex:Sister )
>>> parent = Class(exNs.Parent)
>>> male = Class(exNs.Male)
>>> father = parent & male
>>> father.identifier = exNs.Father
>>> list(father.parents)
[Class: ex:Parent , Class: ex:Male ]
```

serialize(*graph*)

setupNounAnnotations(*nounAnnotations*)

property subClassOf

subSumpteeIds()

class rdflib.extras.infixowl.**ClassNamespaceFactory**
Bases: *rdflib.namespace.Namespace*

__getattr__(*name*)

__getitem__(*key*, *default=None*)
Return self[key].

__module__ = 'rdflib.extras.infixowl'

term(*name*)

rdflib.extras.infixowl.**classOrIdentifier**(*thing*)

rdflib.extras.infixowl.**classOrTerm**(*thing*)

rdflib.extras.infixowl.**CommonNSBindings**(*graph*, *additionalNS={}*)
Takes a graph and binds the common namespaces (rdf,rdfs, & owl)

rdflib.extras.infixowl.**ComponentTerms**(*cls*)
Takes a Class instance and returns a generator over the classes that are involved in its definition, ignoring unnamed classes

rdflib.extras.infixowl.**DeepClassClear**(*classToPrune*)
Recursively clear the given class, continuing where any related class is an anonymous class

```
>>> EX = Namespace('http://example.com/')
>>> namespace_manager = NamespaceManager(Graph())
>>> namespace_manager.bind('ex', EX, override=False)
>>> namespace_manager.bind('owl', OWL_NS, override=False)
>>> g = Graph()
>>> g.namespace_manager = namespace_manager
>>> Individual.factoryGraph = g
>>> classB = Class(EX.B)
>>> classC = Class(EX.C)
>>> classD = Class(EX.D)
>>> classE = Class(EX.E)
>>> classF = Class(EX.F)
>>> anonClass = EX.someProp | some | classD
>>> classF += anonClass
>>> list(anonClass.subClassOf)
[Class: ex:F ]
>>> classA = classE | classF | anonClass
>>> classB += classA
>>> classA.equivalentClass = [Class()]
>>> classB.subClassOf = [EX.someProp | some | classC]
>>> classA
( ex:E OR ex:F OR ( ex:someProp SOME ex:D ) )
>>> DeepClassClear(classA)
>>> classA
(  )
>>> list(anonClass.subClassOf)
[]
>>> classB
Class: ex:B SubClassOf: ( ex:someProp SOME ex:C )
```

```
>>> otherClass = classD | anonClass
>>> otherClass
( ex:D OR ( ex:someProp SOME ex:D ) )
```

(continues on next page)

(continued from previous page)

```
>>> DeepClassClear(otherClass)
>>> otherClass
(  )
>>> otherClass.delete()
>>> list(g.triples((otherClass.identifier, None, None)))
[]
```

class rdflib.extras.infixowl.**EnumeratedClass**(*identifier=None*, *members=None*, *graph=None*)

Bases: *rdflib.extras.infixowl.OWLRDFListProxy*, *rdflib.extras.infixowl.Class*

Class for owl:oneOf forms:

OWL Abstract Syntax is used

axiom ::= 'EnumeratedClass(' classID ['Deprecated'] { annotation } { individualID } ')'

```
>>> exNs = Namespace('http://example.com/')
>>> namespace_manager = NamespaceManager(Graph())
>>> namespace_manager.bind('ex', exNs, override=False)
>>> namespace_manager.bind('owl', OWL_NS, override=False)
>>> g = Graph()
>>> g.namespace_manager = namespace_manager
>>> Individual.factoryGraph = g
>>> ogbujiBros = EnumeratedClass(exNs.ogbujicBros,
...                              members=[exNs.chime,
...                                       exNs.uche,
...                                       exNs.ejike])
>>> ogbujiBros
{ ex:chime ex:uche ex:ejike }
>>> col = Collection(g, first(
...    g.objects(predicate=OWL_NS.oneOf, subject=ogbujiBros.identifier)))
>>> [g.qname(item) for item in col]
[u'ex:chime', u'ex:uche', u'ex:ejike']
>>> print(g.serialize(format='n3'))
@prefix ex: <http://example.com/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:ogbujicBros a owl:Class;
    owl:oneOf ( ex:chime ex:uche ex:ejike ) .
```

__init__(*identifier=None*, *members=None*, *graph=None*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.extras.infixowl'

__repr__()

Returns the Manchester Syntax equivalent for this class

isPrimitive()

serialize(*graph*)

rdflib.extras.infixowl.**generateQName**(*graph*, *uri*)

rdflib.extras.infixowl.**GetIdentifiedClasses**(*graph*)

class rdflib.extras.infixowl.**Individual**(*identifier=None*, *graph=None*)

Bases: object

A typed individual

__dict__ = mappingproxy({'__module__': 'rdflib.extras.infixowl', '__doc__': '\n A ty

__init__(*identifier=None*, *graph=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.extras.infixowl'

__weakref__
list of weak references to the object (if defined)

clearInDegree()

clearOutDegree()

delete()

factoryGraph = <Graph identifier=N1178e5afa09b4c258ea9e79c420f0088 (<class 'rdflib.gra

property identifier

replace(*other*)

property sameAs

serialize(*graph*)

property type

exception rdflib.extras.infixowl.**MalformedClass**(*msg*)
Bases: Exception

__init__(*msg*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.extras.infixowl'

__repr__()
Return repr(self).

__weakref__
list of weak references to the object (if defined)

rdflib.extras.infixowl.**manchesterSyntax**(*thing*, *store*, *boolean=None*, *transientList=False*)
Core serialization

class rdflib.extras.infixowl.**Ontology**(*identifier=None*, *imports=None*, *comment=None*, *graph=None*)
Bases: *rdflib.extras.infixowl.AnnotatableTerms*

The owl ontology metadata

__init__(*identifier=None*, *imports=None*, *comment=None*, *graph=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.extras.infixowl'

property imports

setVersion(*version*)

class rdflib.extras.infixowl.**OWLRDFListProxy**(*rdfList*, *members=None*, *graph=None*)
Bases: object

__contains__(*item*)

__delitem__(*key*)

__dict__ = mappingproxy({'__module__': 'rdflib.extras.infixowl', '__init__': <functi

__eq__(*other*)
Equivalence of boolean class constructors is determined by equivalence of its members

__getitem__(*key*)

__hash__ = None

__iadd__(*other*)

__init__(*rdfList*, *members=None*, *graph=None*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

__module__ = 'rdflib.extras.infixowl'

__setitem__(*key*, *value*)

__weakref__
list of weak references to the object (if defined)

append(*item*)

clear()

index(*item*)

class rdflib.extras.infixowl.**Property**(*identifier=None*, *graph=None*, *baseType=rdflib.term.URIRef('http://www.w3.org/2002/07/owl#ObjectProperty')*, *subPropertyOf=None*, *domain=None*, *range=None*, *inverseOf=None*, *otherType=None*, *equivalentProperty=None*, *comment=None*, *verbAnnotations=None*, *nameAnnotation=None*, *nameIsLabel=False*)

Bases: *rdflib.extras.infixowl.AnnotatableTerms*

axiom ::= 'DatatypeProperty(' datavaluedPropertyID ['Deprecated']

{ annotation } { 'super(' datavaluedPropertyID ')'} ['Functional'] { 'domain(' description ')' } { 'range(' dataRange ')' } ')'

'ObjectProperty(' individualvaluedPropertyID ['Deprecated'] { annotation } { 'super(' individualvaluedPropertyID ')' } ['inverseOf(' individualvaluedPropertyID ')'] ['Symmetric'] ['Functional' | 'InverseFunctional' | 'Functional' 'InverseFunctional' | 'Transitive'] { 'domain(' description ')' } { 'range(' description ')' } ')

__init__(*identifier=None*, *graph=None*, *baseType=rdflib.term.URIRef('http://www.w3.org/2002/07/owl#ObjectProperty')*, *subPropertyOf=None*, *domain=None*, *range=None*, *inverseOf=None*, *otherType=None*, *equivalentProperty=None*, *comment=None*, *verbAnnotations=None*, *nameAnnotation=None*, *nameIsLabel=False*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.extras.infixowl'

__repr__()
Return repr(self).

property domain

property extent

property inverseOf

property range

replace(*other*)

serialize(*graph*)

setupVerbAnnotations(*verbAnnotations*)

property subPropertyOf

rdflib.extras.infixowl.**propertyOrIdentifier**(*thing*)

class rdflib.extras.infixowl.**Restriction**(*onProperty*, *graph=<Graph identifier=N079d6936ab5f4bd282e0d5cd8f708641 (<class 'rdflib.graph.Graph'>)>*, *allValuesFrom=None*, *someValuesFrom=None*, *value=None*, *cardinality=None*, *maxCardinality=None*, *minCardinality=None*, *identifier=None*)

Bases: *rdflib.extras.infixowl.Class*

restriction ::= 'restriction('

datavaluedPropertyID dataRestrictionComponent { dataRestrictionComponent } ')'

'restriction(' individualvaluedPropertyID individualRestrictionComponent { individualRestrictionComponent } ')'

__eq__(*other*)

Equivalence of restrictions is determined by equivalence of the property in question and the restriction 'range'

__hash__()

```
>>> b=Class(OWL_NS.Restriction)
>>> c=Class(OWL_NS.Restriction)
>>> len(set([b,c]))
1
```

__init__(*onProperty*, *graph=<Graph identifier=N079d6936ab5f4bd282e0d5cd8f708641 (<class 'rdflib.graph.Graph'>)>*, *allValuesFrom=None*, *someValuesFrom=None*, *value=None*, *cardinality=None*, *maxCardinality=None*, *minCardinality=None*, *identifier=None*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.extras.infixowl'

__repr__()

Returns the Manchester Syntax equivalent for this restriction

property allValuesFrom

property cardinality

property hasValue

isPrimitive()

property maxCardinality

property minCardinality

property onProperty

restrictionKind()

restrictionKinds = [rdflib.term.URIRef('http://www.w3.org/2002/07/owl#allValuesFrom'),

serialize(*graph*)

```
>>> g1 = Graph()
>>> g2 = Graph()
>>> EX = Namespace("http://example.com/")
>>> namespace_manager = NamespaceManager(g1)
>>> namespace_manager.bind('ex', EX, override=False)
>>> namespace_manager = NamespaceManager(g2)
>>> namespace_manager.bind('ex', EX, override=False)
>>> Individual.factoryGraph = g1
>>> prop = Property(EX.someProp, baseType=OWL_NS.DatatypeProperty)
>>> restr1 = (Property(
...     EX.someProp,
...     baseType=OWL_NS.DatatypeProperty)) | some | (Class(EX.Foo))
>>> restr1
( ex:someProp SOME ex:Foo )
>>> restr1.serialize(g2)
>>> Individual.factoryGraph = g2
>>> list(Property(
...      EX.someProp,baseType=None).type
... )
[rdflib.term.URIRef(
    u'http://www.w3.org/2002/07/owl#DatatypeProperty')]
```

property someValuesFrom

rdflib.extras.infixowl.**termDeletionDecorator**(*prop*)

Module contents

rdflib.plugins package

Subpackages

rdflib.plugins.parsers package

Submodules

rdflib.plugins.parsers.notation3 module

notation3.py - Standalone Notation3 Parser Derived from CWM, the Closed World Machine

Authors of the original suite:

- Dan Connolly <@@>
- Tim Berners-Lee <@@>

- Yosi Scharf <@@>
- Joseph M. Reagle Jr. <reagle@w3.org>
- Rich Salz <rsalz@zolera.com>

http://www.w3.org/2000/10/swap/notation3.py

exception rdflib.plugins.parsers.notation3.**BadSyntax**(*uri*, *lines*, *argstr*, *i*, *why*)
Bases: SyntaxError

__init__(*uri*, *lines*, *argstr*, *i*, *why*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.notation3'

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

property message

class rdflib.plugins.parsers.notation3.**N3Parser**
Bases: *rdflib.plugins.parsers.notation3.TurtleParser*

An RDFLib parser for Notation3

See http://www.w3.org/DesignIssues/Notation3.html

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.notation3'

parse(*source*, *graph*, *encoding='utf-8'*)

class rdflib.plugins.parsers.notation3.**TurtleParser**
Bases: *rdflib.parser.Parser*

An RDFLib parser for Turtle

See http://www.w3.org/TR/turtle/

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.notation3'

parse(*source*, *graph*, *encoding='utf-8'*, *turtle=True*)

rdflib.plugins.parsers.notation3.**splitFragP**(*uriref*, *punct=0*)
split a URI reference before the fragment

Punctuation is kept.

e.g.

```
>>> splitFragP("abc#def")
('abc', '#def')
```

```
>>> splitFragP("abcdef")
('abcdef', '')
```

rdflib.plugins.parsers.notation3.**join**(*here*, *there*)
join an absolute URI and URI reference (non-ascii characters are supported/doctested; haven't checked the details of the IRI spec though)

`here` is assumed to be absolute. `there` is URI reference.

```
>>> join('http://example/x/y/z', '../abc')
'http://example/x/abc'
```

Raise ValueError if there uses relative path syntax but here has no hierarchical path.

```
>>> join('mid:foo@example', '../foo')
Traceback (most recent call last):
    raise ValueError(here)
ValueError: Base <mid:foo@example> has no slash
after colon - with relative '../foo'.
```

```
>>> join('http://example/x/y/z', '')
'http://example/x/y/z'
```

```
>>> join('mid:foo@example', '#foo')
'mid:foo@example#foo'
```

We grok IRIs

```
>>> len(u'Andr\xe9')
5
```

```
>>> join('http://example.org/', u'#Andr\xe9')
u'http://example.org/#Andr\xe9'
```

rdflib.plugins.parsers.notation3.**base**()
The base URI for this process - the Web equiv of cwd

Relative or abolute unix-standard filenames parsed relative to this yeild the URI of the file. If we had a reliable way of getting a computer name, we should put it in the hostname just to prevent ambiguity

rdflib.plugins.parsers.notation3.**runNamespace**()
Returns a URI suitable as a namespace for run-local objects

rdflib.plugins.parsers.notation3.**uniqueURI**()
A unique URI

rdflib.plugins.parsers.notation3.**hexify**(*ustr*)
Use URL encoding to return an ASCII string corresponding to the given UTF8 string

```
>>> hexify("http://example/a b")
%(b)s'http://example/a%%20b'
```

rdflib.plugins.parsers.nquads module

This is a rdflib plugin for parsing NQuad files into Conjunctive graphs that can be used and queried. The store that backs the graph *must* be able to handle contexts.

```
>>> from rdflib import ConjunctiveGraph, URIRef, Namespace
>>> g = ConjunctiveGraph()
>>> data = open("test/nquads.rdflib/example.nquads", "rb")
>>> g.parse(data, format="nquads")
<Graph identifier=... (<class 'rdflib.graph.Graph'>)>
>>> assert len(g.store) == 449
>>> # There should be 16 separate contexts
>>> assert len([x for x in g.store.contexts()]) == 16
>>> # is the name of entity E10009 "Arco Publications"?
>>> #   (in graph http://bibliographica.org/entity/E10009)
>>> # Looking for:
>>> # <http://bibliographica.org/entity/E10009>
>>> #   <http://xmlns.com/foaf/0.1/name>
>>> #   "Arco Publications"
>>> #   <http://bibliographica.org/entity/E10009>
>>> s = URIRef("http://bibliographica.org/entity/E10009")
>>> FOAF = Namespace("http://xmlns.com/foaf/0.1/")
>>> assert(g.value(s, FOAF.name).eq("Arco Publications"))
```

class rdflib.plugins.parsers.nquads.**NQuadsParser**(*sink=None*)

Bases: *rdflib.plugins.parsers.ntriples.NTriplesParser*

__module__ = 'rdflib.plugins.parsers.nquads'

parse(*inputsource*, *sink*, ***kwargs*)

Parse f as an N-Triples file.

parseline()

rdflib.plugins.parsers.nt module

class rdflib.plugins.parsers.nt.**NTSink**(*graph*)

Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.parsers.nt', '__init__': <fun

__init__(*graph*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.nt'

__weakref__

list of weak references to the object (if defined)

triple(*s*, *p*, *o*)

class rdflib.plugins.parsers.nt.**NTParser**

Bases: *rdflib.parser.Parser*

parser for the ntriples format, often stored with the .nt extension

See http://www.w3.org/TR/rdf-testcases/#ntriples

__init__()

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.nt'

parse(*source*, *sink*, *baseURI=None*)

rdflib.plugins.parsers.ntriples module

N-Triples Parser License: GPL 2, W3C, BSD, or MIT Author: Sean B. Palmer, inamidst.com

rdflib.plugins.parsers.ntriples.**unquote**(*s*)
Unquote an N-Triples string.

rdflib.plugins.parsers.ntriples.**uriquote**(*uri*)

class rdflib.plugins.parsers.ntriples.**Sink**
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.parsers.ntriples', '__init__':

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.ntriples'

__weakref__
list of weak references to the object (if defined)

triple(*s*, *p*, *o*)

class rdflib.plugins.parsers.ntriples.**NTriplesParser**(*sink=None*)
Bases: *object*

An N-Triples Parser.

Usage:

```
p = NTriplesParser(sink=MySink())
sink = p.parse(f) # file; use parsestring for a string
```

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.parsers.ntriples', '__doc__':

__init__(*sink=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.ntriples'

__weakref__
list of weak references to the object (if defined)

eat(*pattern*)

literal()

nodeid()

object()

parse(*f*)
Parse f as an N-Triples file.

parseline()

parsestring(*s*)
Parse s as an N-Triples string.

peek(*token*)

predicate()

readline()
Read an N-Triples line from buffered input.

subject()

uriref()

rdflib.plugins.parsers.rdfxml module

An RDF/XML parser for RDFLib

rdflib.plugins.parsers.rdfxml.**create_parser**(*target*, *store*)

class rdflib.plugins.parsers.rdfxml.**BagID**(*val*)
Bases: *rdflib.term.URIRef*

__init__(*val*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.rdfxml'

__slots__ = ['li']

li

next_li()

class rdflib.plugins.parsers.rdfxml.**ElementHandler**
Bases: *object*

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.rdfxml'

__slots__ = ['start', 'char', 'end', 'li', 'id', 'base', 'subject', 'predicate', 'obje

base

char

data

datatype

declared

end

id

language

li

list

next_li()

object

predicate

start

subject

class rdflib.plugins.parsers.rdfxml.**RDFXMLHandler**(*store*)
Bases: xml.sax.handler.ContentHandler

__init__(*store*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.rdfxml'

absolutize(*uri*)

add_reified(*sid*, *spo*)

characters(*content*)
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

convert(*name*, *qname*, *attrs*)

property current

document_element_start(*name*, *qname*, *attrs*)

endElementNS(*name*, *qname*)
Signals the end of an element in namespace mode.

The name parameter contains the name of the element type, just as with the startElementNS event.

endPrefixMapping(*prefix*)
End the scope of a prefix-URI mapping.

See startPrefixMapping for details. This event will always occur after the corresponding endElement event, but the order of endPrefixMapping events is not otherwise guaranteed.

error(*message*)

get_current()

get_next()

get_parent()

ignorableWhitespace(*content*)
Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

list_node_element_end(*name*, *qname*)

literal_element_char(*data*)

literal_element_end(*name*, *qname*)

literal_element_start(*name*, *qname*, *attrs*)

property next

node_element_end(*name*, *qname*)

node_element_start(*name*, *qname*, *attrs*)

property parent

processingInstruction(*target*, *data*)
Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

property_element_char(*data*)

property_element_end(*name*, *qname*)

property_element_start(*name*, *qname*, *attrs*)

reset()

setDocumentLocator(*locator*)
Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

startDocument()
Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for setDocumentLocator).

startElementNS(*name*, *qname*, *attrs*)
Signals the start of an element in namespace mode.

The name parameter contains the name of the element type as a (uri, localname) tuple, the qname parameter the raw XML 1.0 name used in the source document, and the attrs parameter holds an instance of the Attributes class containing the attributes of the element.

The uri part of the name tuple is None for elements which have no namespace.

startPrefixMapping(*prefix*, *namespace*)
Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the http://xml.org/sax/features/namespaces feature is true (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the start/endPrefixMapping event supplies the information to the application to expand prefixes in those contexts itself, if necessary.

Note that start/endPrefixMapping events are not guaranteed to be properly nested relative to each-other: all startPrefixMapping events will occur before the corresponding startElement event, and all endPrefixMapping events will occur after the corresponding endElement event, but their order is not guaranteed.

class rdflib.plugins.parsers.rdfxml.**RDFXMLParser**
Bases: *rdflib.parser.Parser*

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.rdfxml'

parse(*source*, *sink*, ***args*)

rdflib.plugins.parsers.trig module

class rdflib.plugins.parsers.trig.**TrigParser**
Bases: *rdflib.parser.Parser*

An RDFLib parser for TriG

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.trig'

parse(*source*, *graph*, *encoding='utf-8'*)

class rdflib.plugins.parsers.trig.**TrigSinkParser**(*store*, *openFormula=None*, *thisDoc=''*, *baseURI=None*, *genPrefix=''*, *why=None*, *turtle=False*)
Bases: rdflib.plugins.parsers.notation3.SinkParser

__module__ = 'rdflib.plugins.parsers.trig'

directiveOrStatement(*argstr*, *h*)

graph(*argstr*, *i*)
Parse trig graph, i.e.

<urn:graphname> = { .. triples .. }

return -1 if it doesn't look like a graph-decl raise Exception if it looks like a graph, but isn't.

labelOrSubject(*argstr*, *i*, *res*)

rdflib.plugins.parsers.trig.**becauseSubGraph**(**args*, ***kwargs*)

rdflib.plugins.parsers.trix module

A TriX parser for RDFLib

rdflib.plugins.parsers.trix.**create_parser**(*store*)

class rdflib.plugins.parsers.trix.**TriXHandler**(*store*)
Bases: xml.sax.handler.ContentHandler

An Sax Handler for TriX. See http://sw.nokia.com/trix/

__init__(*store*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.trix'

characters(*content*)
Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

endElementNS(*name*, *qname*)
Signals the end of an element in namespace mode.

The name parameter contains the name of the element type, just as with the startElementNS event.

endPrefixMapping(*prefix*)
End the scope of a prefix-URI mapping.

See startPrefixMapping for details. This event will always occur after the corresponding endElement event, but the order of endPrefixMapping events is not otherwise guaranteed.

error(*message*)

get_bnode(*label*)

ignorableWhitespace(*content*)
Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

processingInstruction(*target*, *data*)
Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

reset()

setDocumentLocator(*locator*)
Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

startDocument()
Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for setDocumentLocator).

startElementNS(*name*, *qname*, *attrs*)
Signals the start of an element in namespace mode.

The name parameter contains the name of the element type as a (uri, localname) tuple, the qname parameter the raw XML 1.0 name used in the source document, and the attrs parameter holds an instance of the Attributes class containing the attributes of the element.

The uri part of the name tuple is None for elements which have no namespace.

startPrefixMapping(*prefix*, *namespace*)
Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the http://xml.org/sax/features/namespaces feature is true (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the start/endPrefixMapping event supplies the information to the application to expand prefixes in those contexts itself, if necessary.

Note that start/endPrefixMapping events are not guaranteed to be properly nested relative to each-other: all startPrefixMapping events will occur before the corresponding startElement event, and all endPrefixMapping events will occur after the corresponding endElement event, but their order is not guaranteed.

class rdflib.plugins.parsers.trix.**TriXParser**
Bases: *rdflib.parser.Parser*

A parser for TriX. See http://sw.nokia.com/trix/

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.parsers.trix'

parse(*source*, *sink*, ***args*)

Module contents

rdflib.plugins.serializers package

Submodules

rdflib.plugins.serializers.n3 module

Notation 3 (N3) RDF graph serializer for RDFLib.

class rdflib.plugins.serializers.n3.**N3Serializer**(*store*, *parent=None*)
Bases: *rdflib.plugins.serializers.turtle.TurtleSerializer*

__init__(*store*, *parent=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.serializers.n3'

endDocument()

getQName(*uri*, *gen_prefix=True*)

indent(*modifier=0*)
Returns indent string multiplied by the depth

isDone(*subject*)
Return true if subject is serialized

p_clause(*node*, *position*)

path(*node*, *position*, *newline=False*)

preprocessTriple(*triple*)

reset()

s_clause(*subject*)

short_name = 'n3'

startDocument()

statement(*subject*)

subjectDone(*subject*)
Mark a subject as done.

rdflib.plugins.serializers.nquads module

class rdflib.plugins.serializers.nquads.**NQuadsSerializer**(*store*)
Bases: *rdflib.serializer.Serializer*

__init__(*store*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.serializers.nquads'

serialize(*stream*, *base=None*, *encoding=None*, ***args*)
Abstract method

rdflib.plugins.serializers.nt module

N-Triples RDF graph serializer for RDFLib. See <http://www.w3.org/TR/rdf-testcases/#ntriples> for details about the format.

class rdflib.plugins.serializers.nt.**NTSerializer**(*store*)
Bases: *rdflib.serializer.Serializer*

Serializes RDF graphs to NTriples format.

__init__(*store*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.serializers.nt'

serialize(*stream*, *base=None*, *encoding=None*, ***args*)
Abstract method

rdflib.plugins.serializers.rdfxml module

rdflib.plugins.serializers.rdfxml.**fix**(*val*)
strip off _: from nodeIDs... as they are not valid NCNames

class rdflib.plugins.serializers.rdfxml.**XMLSerializer**(*store*)
Bases: *rdflib.serializer.Serializer*

__init__(*store*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.serializers.rdfxml'

predicate(*predicate*, *object*, *depth=1*)

serialize(*stream*, *base=None*, *encoding=None*, ***args*)
Abstract method

subject(*subject*, *depth=1*)

class rdflib.plugins.serializers.rdfxml.**PrettyXMLSerializer**(*store*, *max_depth=3*)
Bases: *rdflib.serializer.Serializer*

__init__(*store*, *max_depth=3*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.serializers.rdfxml'

predicate(*predicate*, *object*, *depth=1*)

serialize(*stream*, *base=None*, *encoding=None*, ***args*)
Abstract method

subject(*subject*, *depth=1*)

rdflib.plugins.serializers.trig module

Trig RDF graph serializer for RDFLib. See <http://www.w3.org/TR/trig/> for syntax specification.

class rdflib.plugins.serializers.trig.**TrigSerializer**(*store*)
Bases: *rdflib.plugins.serializers.turtle.TurtleSerializer*

__init__(*store*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.serializers.trig'

indentString = ' '

preprocess()

reset()

serialize(*stream*, *base=None*, *encoding=None*, *spacious=None*, ***args*)
Abstract method

short_name = 'trig'

rdflib.plugins.serializers.trix module

class rdflib.plugins.serializers.trix.**TriXSerializer**(*store*)
Bases: *rdflib.serializer.Serializer*

__init__(*store*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.serializers.trix'

serialize(*stream*, *base=None*, *encoding=None*, ***args*)
Abstract method

rdflib.plugins.serializers.turtle module

Turtle RDF graph serializer for RDFLib. See <http://www.w3.org/TeamSubmission/turtle/> for syntax specification.

class rdflib.plugins.serializers.turtle.**RecursiveSerializer**(*store*)
Bases: *rdflib.serializer.Serializer*

__init__(*store*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.serializers.turtle'

addNamespace(*prefix*, *uri*)

buildPredicateHash(*subject*)
Build a hash key by predicate to a list of objects for the given subject

checkSubject(*subject*)
Check to see if the subject should be serialized yet

indent(*modifier=0*)
Returns indent string multiplied by the depth

indentString = ' '

isDone(*subject*)
Return true if subject is serialized

maxDepth = 10

orderSubjects()

predicateOrder = [rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#type'

preprocess()

preprocessTriple(*spo*)

reset()

roundtrip_prefixes = ()

sortProperties(*properties*)
Take a hash from predicate uris to lists of values. Sort the lists of values. Return a sorted list of properties.

subjectDone(*subject*)
Mark a subject as done.

topClasses = [rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#Class')]

write(*text*)
Write text in given encoding.

class rdflib.plugins.serializers.turtle.**TurtleSerializer**(*store*)
Bases: *rdflib.plugins.serializers.turtle.RecursiveSerializer*

__init__(*store*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.serializers.turtle'

addNamespace(*prefix*, *namespace*)

doList(*l_*)

endDocument()

getQName(*uri*, *gen_prefix=True*)

indentString = ' '

isValidList(*l_*)
Checks if l is a valid RDF list, i.e. no nodes have other properties.

label(*node*, *position*)

objectList(*objects*)

p_default(*node*, *position*, *newline=False*)

p_squared(*node*, *position*, *newline=False*)

path(*node*, *position*, *newline=False*)

predicateList(*subject*, *newline=False*)

preprocessTriple(*triple*)

reset()

s_default(*subject*)

s_squared(*subject*)

serialize(*stream*, *base=None*, *encoding=None*, *spacious=None*, ***args*)
Abstract method

short_name = 'turtle'

startDocument()

statement(*subject*)

verb(*node*, *newline=False*)

rdflib.plugins.serializers.xmlwriter module

class rdflib.plugins.serializers.xmlwriter.**XMLWriter**(*stream*, *namespace_manager*, *encoding=None*, *decl=1*, *extra_ns=None*)
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.serializers.xmlwriter', '__ini

__init__(*stream*, *namespace_manager*, *encoding=None*, *decl=1*, *extra_ns=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.serializers.xmlwriter'

__weakref__
list of weak references to the object (if defined)

attribute(*uri*, *value*)

element(*uri*, *content*, *attributes={}*)
Utility method for adding a complete simple element

property indent

namespaces(*namespaces=None*)

pop(*uri=None*)

push(*uri*)

qname(*uri*)
Compute qname for a uri using our extra namespaces, or the given namespace manager

text(*text*)

Module contents

rdflib.plugins.sparql package

Subpackages

rdflib.plugins.sparql.results package

Submodules

rdflib.plugins.sparql.results.csvresults module

This module implements a parser and serializer for the CSV SPARQL result formats

http://www.w3.org/TR/sparql11-results-csv-tsv/

class rdflib.plugins.sparql.results.csvresults.**CSVResultParser**
Bases: *rdflib.query.ResultParser*

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.results.csvresults'

convertTerm(*t*)

parse(*source*, *content_type=None*)
return a Result object

parseRow(*row*, *v*)

class rdflib.plugins.sparql.results.csvresults.**CSVResultSerializer**(*result*)
Bases: *rdflib.query.ResultSerializer*

__init__(*result*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.results.csvresults'

serialize(*stream*, *encoding='utf-8'*, ***kwargs*)
return a string properly serialized

serializeTerm(*term*, *encoding*)

rdflib.plugins.sparql.results.graph module

class rdflib.plugins.sparql.results.graph.**GraphResultParser**
Bases: *rdflib.query.ResultParser*

__module__ = 'rdflib.plugins.sparql.results.graph'

parse(*source*, *content_type*)
return a Result object

rdflib.plugins.sparql.results.jsonresults module

class rdflib.plugins.sparql.results.jsonresults.**JSONResult**(*json*)
Bases: *rdflib.query.Result*

__init__(*json*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.results.jsonresults'

class rdflib.plugins.sparql.results.jsonresults.**JSONResultParser**
Bases: *rdflib.query.ResultParser*

__module__ = 'rdflib.plugins.sparql.results.jsonresults'

parse(*source*, *content_type=None*)
return a Result object

class rdflib.plugins.sparql.results.jsonresults.**JSONResultSerializer**(*result*)
Bases: *rdflib.query.ResultSerializer*

__init__(*result*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.results.jsonresults'

serialize(*stream*, *encoding=None*)
return a string properly serialized

rdflib.plugins.sparql.results.jsonresults.**parseJsonTerm**(*d*)
rdflib object (Literal, URIRef, BNode) for the given json-format dict.

input is like: { 'type': 'uri', 'value': 'http://famegame.com/2006/01/username' } { 'type': 'literal', 'value': 'drewp' }

rdflib.plugins.sparql.results.jsonresults.**termToJSON**(*self*, *term*)

rdflib.plugins.sparql.results.rdfresults module

class rdflib.plugins.sparql.results.rdfresults.**RDFResult**(*source*, ***kwargs*)
Bases: *rdflib.query.Result*

__init__(*source*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.results.rdfresults'

class rdflib.plugins.sparql.results.rdfresults.**RDFResultParser**
Bases: *rdflib.query.ResultParser*

__module__ = 'rdflib.plugins.sparql.results.rdfresults'

parse(*source*, ***kwargs*)
return a Result object

rdflib.plugins.sparql.results.tsvresults module

This implements the Tab Separated SPARQL Result Format

It is implemented with pyparsing, reusing the elements from the SPARQL Parser

class rdflib.plugins.sparql.results.tsvresults.**TSVResultParser**
Bases: *rdflib.query.ResultParser*

__module__ = 'rdflib.plugins.sparql.results.tsvresults'

convertTerm(*t*)

parse(*source*, *content_type=None*)
return a Result object

rdflib.plugins.sparql.results.txtresults module

class rdflib.plugins.sparql.results.txtresults.**TXTResultSerializer**(*result*)
Bases: *rdflib.query.ResultSerializer*

A write only QueryResult serializer for text/ascii tables

__module__ = 'rdflib.plugins.sparql.results.txtresults'

serialize(*stream*, *encoding*, *namespace_manager=None*)
return a text table of query results

rdflib.plugins.sparql.results.xmlresults module

class rdflib.plugins.sparql.results.xmlresults.**SPARQLXMLWriter**(*output*, *encoding='utf-8'*)
Bases: object

Python saxutils-based SPARQL XML Writer

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.sparql.results.xmlresults', '_

__init__(*output*, *encoding='utf-8'*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.results.xmlresults'

__weakref__
list of weak references to the object (if defined)

close()

write_ask(*val*)

write_binding(*name*, *val*)

write_end_result()

write_header(*allvarsL*)

write_results_header()

write_start_result()

class rdflib.plugins.sparql.results.xmlresults.**XMLResult**(*source*, *content_type=None*)
Bases: *rdflib.query.Result*

__init__(*source*, *content_type=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.results.xmlresults'

class rdflib.plugins.sparql.results.xmlresults.**XMLResultParser**
Bases: *rdflib.query.ResultParser*

__module__ = 'rdflib.plugins.sparql.results.xmlresults'

parse(*source*, *content_type=None*)
return a Result object

class rdflib.plugins.sparql.results.xmlresults.**XMLResultSerializer**(*result*)
Bases: *rdflib.query.ResultSerializer*

__init__(*result*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.results.xmlresults'

serialize(*stream*, *encoding='utf-8'*)
return a string properly serialized

rdflib.plugins.sparql.results.xmlresults.**log = <Logger rdflib.plugins.sparql.results.xmlres**
A Parser for SPARQL results in XML:

http://www.w3.org/TR/rdf-sparql-XMLres/

Bits and pieces borrowed from: http://projects.bigasterisk.com/sparqlhttp/

Authors: Drew Perttula, Gunnar Aastrand Grimnes

rdflib.plugins.sparql.results.xmlresults.**parseTerm**(*element*)
rdflib object (Literal, URIRef, BNode) for the given elementtree element

Module contents

Parsers and serializers for SPARQL Result formats

Submodules

rdflib.plugins.sparql.aggregates module

class rdflib.plugins.sparql.aggregates.**Accumulator**(*aggregation*)
Bases: object

abstract base class for different aggregation functions

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.sparql.aggregates', '__doc__':

__init__(*aggregation*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.aggregates'

__weakref__
list of weak references to the object (if defined)

dont_care(*row*)
skips distinct test

set_value(*bindings*)
sets final value in bindings

use_row(*row*)
tests distinct with set

class rdflib.plugins.sparql.aggregates.**Aggregator**(*aggregations*)
Bases: object

combines different Accumulator objects

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.sparql.aggregates', '__doc__':

__init__(*aggregations*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.aggregates'

__weakref__
list of weak references to the object (if defined)

accumulator_classes = {'Aggregate_Avg': <class 'rdflib.plugins.sparql.aggregates.Aver

get_bindings()
calculate and set last values

update(*row*)
update all own accumulators

class rdflib.plugins.sparql.aggregates.**Average**(*aggregation*)
Bases: *rdflib.plugins.sparql.aggregates.Accumulator*

__init__(*aggregation*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.aggregates'

get_value()

update(*row*, *aggregator*)

class rdflib.plugins.sparql.aggregates.**Counter**(*aggregation*)
Bases: *rdflib.plugins.sparql.aggregates.Accumulator*

__init__(*aggregation*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.aggregates'

eval_full_row(*row*)

eval_row(*row*)

get_value()

update(*row*, *aggregator*)

use_row(*row*)
tests distinct with set

class rdflib.plugins.sparql.aggregates.**Extremum**(*aggregation*)
Bases: *rdflib.plugins.sparql.aggregates.Accumulator*

abstract base class for Minimum and Maximum

__init__(*aggregation*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.aggregates'

set_value(*bindings*)
sets final value in bindings

update(*row*, *aggregator*)

class rdflib.plugins.sparql.aggregates.**GroupConcat**(*aggregation*)
Bases: *rdflib.plugins.sparql.aggregates.Accumulator*

__init__(*aggregation*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.aggregates'

get_value()

update(*row*, *aggregator*)

class rdflib.plugins.sparql.aggregates.**Maximum**(*aggregation*)
Bases: *rdflib.plugins.sparql.aggregates.Extremum*

__module__ = 'rdflib.plugins.sparql.aggregates'

compare(*val1*, *val2*)

class rdflib.plugins.sparql.aggregates.**Minimum**(*aggregation*)
Bases: *rdflib.plugins.sparql.aggregates.Extremum*

__module__ = 'rdflib.plugins.sparql.aggregates'

compare(*val1*, *val2*)

class rdflib.plugins.sparql.aggregates.**Sample**(*aggregation*)
Bases: *rdflib.plugins.sparql.aggregates.Accumulator*

takes the first eligable value

__init__(*aggregation*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.aggregates'

get_value()

update(*row*, *aggregator*)

class rdflib.plugins.sparql.aggregates.**Sum**(*aggregation*)
Bases: *rdflib.plugins.sparql.aggregates.Accumulator*

__init__(*aggregation*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.aggregates'

get_value()

update(*row*, *aggregator*)

rdflib.plugins.sparql.aggregates.**type_safe_numbers**(**args*)

rdflib.plugins.sparql.algebra module

Converting the 'parse-tree' output of pyparsing to a SPARQL Algebra expression

http://www.w3.org/TR/sparql11-query/#sparqlQuery

rdflib.plugins.sparql.algebra.**BGP**(*triples=None*)

rdflib.plugins.sparql.algebra.**Extend**(*p*, *expr*, *var*)

rdflib.plugins.sparql.algebra.**Filter**(*expr*, *p*)

rdflib.plugins.sparql.algebra.**Graph**(*term*, *graph*)

rdflib.plugins.sparql.algebra.**Group**(*p*, *expr=None*)

rdflib.plugins.sparql.algebra.**Join**(*p1*, *p2*)

rdflib.plugins.sparql.algebra.**LeftJoin**(*p1*, *p2*, *expr*)

rdflib.plugins.sparql.algebra.**Minus**(*p1*, *p2*)

rdflib.plugins.sparql.algebra.**OrderBy**(*p*, *expr*)

rdflib.plugins.sparql.algebra.**Project**(*p*, *PV*)

exception rdflib.plugins.sparql.algebra.**StopTraversal**(*rv*)
Bases: Exception

__init__(*rv*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.algebra'

__weakref__
list of weak references to the object (if defined)

rdflib.plugins.sparql.algebra.**ToMultiSet**(*p*)

rdflib.plugins.sparql.algebra.**Union**(*p1*, *p2*)

rdflib.plugins.sparql.algebra.**Values**(*res*)

rdflib.plugins.sparql.algebra.**analyse**(*n*, *children*)
Some things can be lazily joined. This propegates whether they can up the tree and sets lazy flags for all joins

rdflib.plugins.sparql.algebra.**collectAndRemoveFilters**(*parts*)
FILTER expressions apply to the whole group graph pattern in which they appear.

http://www.w3.org/TR/sparql11-query/#sparqlCollectFilters

rdflib.plugins.sparql.algebra.**pprintAlgebra**(*q*)

rdflib.plugins.sparql.algebra.**reorderTriples**(*l_*)
Reorder triple patterns so that we execute the ones with most bindings first

rdflib.plugins.sparql.algebra.**simplify**(*n*)
Remove joins to empty BGPs

rdflib.plugins.sparql.algebra.**translate**(*q*)
http://www.w3.org/TR/sparql11-query/#convertSolMod

rdflib.plugins.sparql.algebra.**translateAggregates**(*q*, *M*)

rdflib.plugins.sparql.algebra.**translateExists**(*e*)
Translate the graph pattern used by EXISTS and NOT EXISTS http://www.w3.org/TR/sparql11-query/#sparqlCollectFilters

rdflib.plugins.sparql.algebra.**translateGraphGraphPattern**(*graphPattern*)

rdflib.plugins.sparql.algebra.**translateGroupGraphPattern**(*graphPattern*)
http://www.w3.org/TR/sparql11-query/#convertGraphPattern

rdflib.plugins.sparql.algebra.**translateGroupOrUnionGraphPattern**(*graphPattern*)

rdflib.plugins.sparql.algebra.**translateInlineData**(*graphPattern*)

rdflib.plugins.sparql.algebra.**translatePName**(*p*, *prologue*)
Expand prefixed/relative URIs

rdflib.plugins.sparql.algebra.**translatePath**(*p*)
Translate PropertyPath expressions

rdflib.plugins.sparql.algebra.**translatePrologue**(*p*, *base*, *initNs=None*, *prologue=None*)

rdflib.plugins.sparql.algebra.**translateQuads**(*quads*)

rdflib.plugins.sparql.algebra.**translateQuery**(*q*, *base=None*, *initNs=None*)
Translate a query-parsetree to a SPARQL Algebra Expression

Return a rdflib.plugins.sparql.sparql.Query object

rdflib.plugins.sparql.algebra.**translateUpdate**(*q*, *base=None*, *initNs=None*)
Returns a list of SPARQL Update Algebra expressions

rdflib.plugins.sparql.algebra.**translateUpdate1**(*u*, *prologue*)

rdflib.plugins.sparql.algebra.**translateValues**(*v*)

rdflib.plugins.sparql.algebra.**traverse**(*tree*, *visitPre=<function <lambda>>*, *visitPost=<function <lambda>>*, *complete=None*)
Traverse tree, visit each node with visit function visit function may raise StopTraversal to stop traversal if complete!=None, it is returned on complete traversal, otherwise the transformed tree is returned

rdflib.plugins.sparql.algebra.**triples**(*l*)

rdflib.plugins.sparql.datatypes module

Utility functions for supporting the XML Schema Datatypes hierarchy

rdflib.plugins.sparql.datatypes.**type_promotion**(*t1*, *t2*)

rdflib.plugins.sparql.evaluate module

These method recursively evaluate the SPARQL Algebra

evalQuery is the entry-point, it will setup context and return the SPARQLResult object

evalPart is called on each level and will delegate to the right method

A rdflib.plugins.sparql.sparql.QueryContext is passed along, keeping information needed for evaluation

A list of dicts (solution mappings) is returned, apart from GroupBy which may also return a dict of list of dicts

rdflib.plugins.sparql.evaluate.**evalAggregateJoin**(*ctx*, *agg*)

rdflib.plugins.sparql.evaluate.**evalAskQuery**(*ctx*, *query*)

rdflib.plugins.sparql.evaluate.**evalBGP**(*ctx*, *bgp*)
A basic graph pattern

rdflib.plugins.sparql.evaluate.**evalConstructQuery**(*ctx*, *query*)

rdflib.plugins.sparql.evaluate.**evalDistinct**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalExtend**(*ctx*, *extend*)

rdflib.plugins.sparql.evaluate.**evalFilter**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalGraph**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalGroup**(*ctx*, *group*)
http://www.w3.org/TR/sparql11-query/#defn_algGroup

rdflib.plugins.sparql.evaluate.**evalJoin**(*ctx*, *join*)

rdflib.plugins.sparql.evaluate.**evalLazyJoin**(*ctx*, *join*)
A lazy join will push the variables bound in the first part to the second part, essentially doing the join implicitly hopefully evaluating much fewer triples

rdflib.plugins.sparql.evaluate.**evalLeftJoin**(*ctx*, *join*)

rdflib.plugins.sparql.evaluate.**evalMinus**(*ctx*, *minus*)

rdflib.plugins.sparql.evaluate.**evalMultiset**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalOrderBy**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalPart**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalProject**(*ctx*, *project*)

rdflib.plugins.sparql.evaluate.**evalQuery**(*graph*, *query*, *initBindings*, *base=None*)

rdflib.plugins.sparql.evaluate.**evalReduced**(*ctx*, *part*)
apply REDUCED to result

REDUCED is not as strict as DISTINCT, but if the incoming rows were sorted it should produce the same result with limited extra memory and time per incoming row.

rdflib.plugins.sparql.evaluate.**evalSelectQuery**(*ctx*, *query*)

rdflib.plugins.sparql.evaluate.**evalServiceQuery**(*ctx*, *part*)

rdflib.plugins.sparql.evaluate.**evalSlice**(*ctx*, *slice*)

rdflib.plugins.sparql.evaluate.**evalUnion**(*ctx*, *union*)

rdflib.plugins.sparql.evaluate.**evalValues**(*ctx*, *part*)

rdflib.plugins.sparql.evalutils module

rdflib.plugins.sparql.operators module

This contains evaluation functions for expressions

They get bound as instances-methods to the CompValue objects from parserutils using setEvalFn

rdflib.plugins.sparql.operators.**AdditiveExpression**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_ABS**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-abs

rdflib.plugins.sparql.operators.**Builtin_BNODE**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-bnode

rdflib.plugins.sparql.operators.**Builtin_BOUND**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-bound

rdflib.plugins.sparql.operators.**Builtin_CEIL**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-ceil

rdflib.plugins.sparql.operators.**Builtin_COALESCE**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-coalesce

rdflib.plugins.sparql.operators.**Builtin_CONCAT**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-concat

rdflib.plugins.sparql.operators.**Builtin_CONTAINS**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strcontains

rdflib.plugins.sparql.operators.**Builtin_DATATYPE**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_DAY**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_ENCODE_FOR_URI**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_EXISTS**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_FLOOR**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-floor

rdflib.plugins.sparql.operators.**Builtin_HOURS**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_IF**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-if

rdflib.plugins.sparql.operators.**Builtin_IRI**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-iri

rdflib.plugins.sparql.operators.**Builtin_LANG**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-lang

Returns the language tag of ltrl, if it has one. It returns "" if ltrl has no language tag. Note that the RDF data model does not include literals with an empty language tag.

rdflib.plugins.sparql.operators.**Builtin_LANGMATCHES**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-langMatches

rdflib.plugins.sparql.operators.**Builtin_LCASE**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_MD5**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_MINUTES**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_MONTH**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_NOW**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-now

rdflib.plugins.sparql.operators.**Builtin_RAND**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#idp2133952

rdflib.plugins.sparql.operators.**Builtin_REGEX**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-regex Invokes the XPath fn:matches function to match text against a regular expression pattern. The regular expression language is defined in XQuery 1.0 and XPath 2.0 Functions and Operators section 7.6.1 Regular Expression Syntax

rdflib.plugins.sparql.operators.**Builtin_REPLACE**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-substr

rdflib.plugins.sparql.operators.**Builtin_ROUND**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-round

rdflib.plugins.sparql.operators.**Builtin_SECONDS**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-seconds

rdflib.plugins.sparql.operators.**Builtin_SHA1**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_SHA256**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_SHA384**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_SHA512**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_STR**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_STRAFTER**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strafter

rdflib.plugins.sparql.operators.**Builtin_STRBEFORE**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strbefore

rdflib.plugins.sparql.operators.**Builtin_STRDT**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strdt

rdflib.plugins.sparql.operators.**Builtin_STRENDS**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strends

rdflib.plugins.sparql.operators.**Builtin_STRLANG**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strlang

rdflib.plugins.sparql.operators.**Builtin_STRLEN**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_STRSTARTS**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strstarts

rdflib.plugins.sparql.operators.**Builtin_STRUUID**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strdt

rdflib.plugins.sparql.operators.**Builtin_SUBSTR**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-substr

rdflib.plugins.sparql.operators.**Builtin_TIMEZONE**(*e*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-timezone

Returns the timezone part of arg as an xsd:dayTimeDuration.

Raises an error if there is no timezone.

rdflib.plugins.sparql.operators.**Builtin_TZ**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_UCASE**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_UUID**(*expr*, *ctx*)
http://www.w3.org/TR/sparql11-query/#func-strdt

rdflib.plugins.sparql.operators.**Builtin_YEAR**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_isBLANK**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_isIRI**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_isLITERAL**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_isNUMERIC**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**Builtin_sameTerm**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**ConditionalAndExpression**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**ConditionalOrExpression**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**EBV**(*rt*)

- If the argument is a typed literal with a datatype of xsd:boolean, the EBV is the value of that argument.
- If the argument is a plain literal or a typed literal with a datatype of xsd:string, the EBV is false if the operand value has zero length; otherwise the EBV is true.
- If the argument is a numeric type or a typed literal with a datatype derived from a numeric type, the EBV is false if the operand value is NaN or is numerically equal to zero; otherwise the EBV is true.
- All other arguments, including unbound arguments, produce a type error.

rdflib.plugins.sparql.operators.**Function**(*e*, *ctx*)
Custom functions and casts

rdflib.plugins.sparql.operators.**MultiplicativeExpression**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**RelationalExpression**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**UnaryMinus**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**UnaryNot**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**UnaryPlus**(*expr*, *ctx*)

rdflib.plugins.sparql.operators.**and_**(**args*)

rdflib.plugins.sparql.operators.**custom_function**(*uri*, *override=False*, *raw=False*)
Decorator version of *register_custom_function()*.

rdflib.plugins.sparql.operators.**datetime**(*e*)

rdflib.plugins.sparql.operators.**default_cast**(*e*, *ctx*)

rdflib.plugins.sparql.operators.**literal**(*s*)

rdflib.plugins.sparql.operators.**not_**(*arg*)

rdflib.plugins.sparql.operators.**numeric**(*expr*)
return a number from a literal http://www.w3.org/TR/xpath20/#promotion

or TypeError

rdflib.plugins.sparql.operators.**register_custom_function**(*uri*, *func*, *override=False*, *raw=False*)
Register a custom SPARQL function.

By default, the function will be passed the RDF terms in the argument list. If raw is True, the function will be passed an Expression and a Context.

The function must return an RDF term, or raise a SparqlError.

rdflib.plugins.sparql.operators.**simplify**(*expr*)

rdflib.plugins.sparql.operators.**string**(*s*)
Make sure the passed thing is a string literal i.e. plain literal, xsd:string literal or lang-tagged literal

rdflib.plugins.sparql.operators.**unregister_custom_function**(*uri*, *func*)

rdflib.plugins.sparql.parser module

SPARQL 1.1 Parser

based on pyparsing

rdflib.plugins.sparql.parser.**expandBNodeTriples**(*terms*)
expand [?p ?o] syntax for implicit bnodes

rdflib.plugins.sparql.parser.**expandCollection**(*terms*)
expand (1 2 3) notation for collections

rdflib.plugins.sparql.parser.**expandTriples**(*terms*)
Expand ; and , syntax for repeat predicates, subjects

rdflib.plugins.sparql.parser.**expandUnicodeEscapes**(*q*)
The syntax of the SPARQL Query Language is expressed over code points in Unicode [UNICODE]. The encoding is always UTF-8 [RFC3629]. Unicode code points may also be expressed using an uXXXX (U+0 to U+FFFF) or UXXXXXXXX syntax (for U+10000 onwards) where X is a hexadecimal digit [0-9A-F]

rdflib.plugins.sparql.parser.**neg**(*literal*)

rdflib.plugins.sparql.parser.**parseQuery**(*q*)

rdflib.plugins.sparql.parser.**parseUpdate**(*q*)

rdflib.plugins.sparql.parser.**setDataType**(*terms*)

rdflib.plugins.sparql.parser.**setLanguage**(*terms*)

rdflib.plugins.sparql.parserutils module

class rdflib.plugins.sparql.parserutils.**Comp**(*name*, *expr*)
Bases: pyparsing.TokenConverter

A pyparsing token for grouping together things with a label Any sub-tokens that are not Params will be ignored.

Returns CompValue / Expr objects - depending on whether evalFn is set.

__init__(*name*, *expr*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.parserutils'

__slotnames__ = []

postParse(*instring*, *loc*, *tokenList*)

setEvalFn(*evalfn*)

class rdflib.plugins.sparql.parserutils.**CompValue**(*name*, ***values*)
Bases: collections.OrderedDict

The result of parsing a Comp Any included Params are avaiable as Dict keys or as attributes

__getattr__(*a*)

__getitem__(*a*)
x.__getitem__(y) <==> x[y]

__init__(*name*, ***values*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.parserutils'

__repr__()
Return repr(self).

__str__()
Return str(self).

clone()

get(*a*, *variables=False*, *errors=False*)
Return the value for key if key is in the dictionary, else default.

class rdflib.plugins.sparql.parserutils.**Expr**(*name*, *evalfn=None*, ***values*)
Bases: *rdflib.plugins.sparql.parserutils.CompValue*

A CompValue that is evaluatable

__init__(*name*, *evalfn=None*, ***values*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.parserutils'

eval(*ctx={}*)

class rdflib.plugins.sparql.parserutils.**Param**(*name*, *expr*, *isList=False*)
Bases: pyparsing.TokenConverter

A pyparsing token for labelling a part of the parse-tree if isList is true repeat occurrences of ParamList have their values merged in a list

__init__(*name*, *expr*, *isList=False*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.parserutils'

__slotnames__ = []

postParse2(*tokenList*)

class rdflib.plugins.sparql.parserutils.**ParamList**(*name*, *expr*)
Bases: *rdflib.plugins.sparql.parserutils.Param*

A shortcut for a Param with isList=True

__init__(*name*, *expr*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.parserutils'

class rdflib.plugins.sparql.parserutils.**ParamValue**(*name*, *tokenList*, *isList*)
Bases: object

The result of parsing a Param This just keeps the name/value All cleverness is in the CompValue

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.sparql.parserutils', '__doc__'

__init__(*name*, *tokenList*, *isList*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.parserutils'

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

class rdflib.plugins.sparql.parserutils.**plist**
Bases: list

this is just a list, but we want our own type to check for

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.sparql.parserutils', '__doc__'

__module__ = 'rdflib.plugins.sparql.parserutils'

__weakref__
list of weak references to the object (if defined)

rdflib.plugins.sparql.parserutils.**prettify_parsetree**(*t*, *indent=''*, *depth=0*)

rdflib.plugins.sparql.parserutils.**value**(*ctx*, *val*, *variables=False*, *errors=False*)
utility function for evaluating something...

Variables will be looked up in the context Normally, non-bound vars is an error, set variables=True to return unbound vars

Normally, an error raises the error, set errors=True to return error

rdflib.plugins.sparql.processor module

Code for tying SPARQL Engine into RDFLib

These should be automatically registered with RDFLib

class rdflib.plugins.sparql.processor.**SPARQLProcessor**(*graph*)
Bases: *rdflib.query.Processor*

__init__(*graph*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.processor'

query(*strOrQuery*, *initBindings={}*, *initNs={}*, *base=None*, *DEBUG=False*)
Evaluate a query with the given initial bindings, and initial namespaces. The given base is used to resolve relative URIs in the query and will be overridden by any BASE given in the query.

class rdflib.plugins.sparql.processor.**SPARQLResult**(*res*)
Bases: *rdflib.query.Result*

__init__(*res*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.processor'

class rdflib.plugins.sparql.processor.**SPARQLUpdateProcessor**(*graph*)
Bases: rdflib.query.UpdateProcessor

__init__(*graph*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.processor'

update(*strOrQuery*, *initBindings={}*, *initNs={}*)

rdflib.plugins.sparql.processor.**prepareQuery**(*queryString*, *initNs={}*, *base=None*)
Parse and translate a SPARQL Query

rdflib.plugins.sparql.processor.**processUpdate**(*graph*, *updateString*, *initBindings={}*, *initNs={}*, *base=None*)
Process a SPARQL Update Request returns Nothing on success or raises Exceptions on error

rdflib.plugins.sparql.sparql module

exception rdflib.plugins.sparql.sparql.**AlreadyBound**
Bases: *rdflib.plugins.sparql.sparql.SPARQLError*

Raised when trying to bind a variable that is already bound!

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.sparql'

class rdflib.plugins.sparql.sparql.**Bindings**(*outer=None*, *d=[]*)
Bases: collections.abc.MutableMapping

A single level of a stack of variable-value bindings. Each dict keeps a reference to the dict below it, any failed lookup is propegated back

In python 3.3 this could be a collections.ChainMap

__abstractmethods__ = frozenset({})

__contains__(*key*)

__delitem__(*key*)

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.sparql.sparql', '__doc__': '

__getitem__(*key*)

__init__(*outer=None*, *d=[]*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

__module__ = 'rdflib.plugins.sparql.sparql'

__repr__()
Return repr(self).

__setitem__(*key*, *value*)

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

class rdflib.plugins.sparql.sparql.**FrozenBindings**(*ctx*, **args*, ***kwargs*)
Bases: *rdflib.plugins.sparql.sparql.FrozenDict*

__abstractmethods__ = frozenset({})

__getitem__(*key*)

__init__(*ctx*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.sparql'

property bnodes

forget(*before*, *_except=None*)
return a frozen dict only of bindings made in self since before

merge(*other*)

property now

project(*vars*)

property prologue

remember(*these*)
return a frozen dict only of bindings in these

class rdflib.plugins.sparql.sparql.**FrozenDict**(**args*, ***kwargs*)
Bases: collections.abc.Mapping

An immutable hashable dict

Taken from http://stackoverflow.com/a/2704866/81121

__abstractmethods__ = frozenset({})

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.sparql.sparql', '__doc__': '

__getitem__(*key*)

__hash__()
Return hash(self).

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

__module__ = 'rdflib.plugins.sparql.sparql'

__repr__()
Return repr(self).

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

compatible(*other*)

disjointDomain(*other*)

merge(*other*)

project(*vars*)

exception rdflib.plugins.sparql.sparql.**NotBoundError**(*msg=None*)
Bases: *rdflib.plugins.sparql.sparql.SPARQLError*

__init__(*msg=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.sparql'

class rdflib.plugins.sparql.sparql.**Prologue**
Bases: object

A class for holding prefixing bindings and base URI information

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.sparql.sparql', '__doc__': '

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.sparql'

__weakref__
list of weak references to the object (if defined)

absolutize(*iri*)
Apply BASE / PREFIXes to URIs (and to datatypes in Literals)

TODO: Move resolving URIs to pre-processing

bind(*prefix*, *uri*)

resolvePName(*prefix*, *localname*)

class rdflib.plugins.sparql.sparql.**Query**(*prologue*, *algebra*)
Bases: object

A parsed and translated query

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.sparql.sparql', '__doc__': '

__init__(*prologue*, *algebra*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.sparql'

__weakref__
list of weak references to the object (if defined)

class rdflib.plugins.sparql.sparql.**QueryContext**(*graph=None*, *bindings=None*, *initBindings=None*)
Bases: object

Query context - passed along when evaluating the query

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.sparql.sparql', '__doc__': '

__getitem__(*key*)

__init__(*graph=None*, *bindings=None*, *initBindings=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.sparql'

__setitem__(*key*, *value*)

__weakref__
list of weak references to the object (if defined)

clean()

clone(*bindings=None*)

property dataset
current dataset

get(*key*, *default=None*)

load(*source*, *default=False*, ***kwargs*)

push()

pushGraph(*graph*)

solution(*vars=None*)
Return a static copy of the current variable bindings as dict

thaw(*frozenbindings*)
Create a new read/write query context from the given solution

exception rdflib.plugins.sparql.sparql.**SPARQLError**(*msg=None*)
Bases: Exception

__init__(*msg=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.sparql'

__weakref__
list of weak references to the object (if defined)

exception rdflib.plugins.sparql.sparql.**SPARQLTypeError**(*msg*)
Bases: *rdflib.plugins.sparql.sparql.SPARQLError*

__init__(*msg*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.sparql.sparql'

rdflib.plugins.sparql.update module

Code for carrying out Update Operations

rdflib.plugins.sparql.update.**evalAdd**(*ctx*, *u*)
add all triples from src to dst

http://www.w3.org/TR/sparql11-update/#add

rdflib.plugins.sparql.update.**evalClear**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#clear

rdflib.plugins.sparql.update.**evalCopy**(*ctx*, *u*)
remove all triples from dst add all triples from src to dst

http://www.w3.org/TR/sparql11-update/#copy

rdflib.plugins.sparql.update.**evalCreate**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#create

rdflib.plugins.sparql.update.**evalDeleteData**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#deleteData

rdflib.plugins.sparql.update.**evalDeleteWhere**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#deleteWhere

rdflib.plugins.sparql.update.**evalDrop**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#drop

rdflib.plugins.sparql.update.**evalInsertData**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#insertData

rdflib.plugins.sparql.update.**evalLoad**(*ctx*, *u*)
http://www.w3.org/TR/sparql11-update/#load

rdflib.plugins.sparql.update.**evalModify**(*ctx*, *u*)

rdflib.plugins.sparql.update.**evalMove**(*ctx*, *u*)
remove all triples from dst add all triples from src to dst remove all triples from src

http://www.w3.org/TR/sparql11-update/#move

rdflib.plugins.sparql.update.**evalUpdate**(*graph*, *update*, *initBindings={}*)
http://www.w3.org/TR/sparql11-update/#updateLanguage

'A request is a sequence of operations [...] Implementations MUST ensure that operations of a single request are executed in a fashion that guarantees the same effects as executing them in lexical order.

Operations all result either in success or failure.

If multiple operations are present in a single request, then a result of failure from any operation MUST abort the sequence of operations, causing the subsequent operations to be ignored.'

This will return None on success and raise Exceptions on error

Module contents

SPARQL implementation for RDFLib

New in version 4.0.

rdflib.plugins.sparql.**CUSTOM_EVALS = {}**
Custom evaluation functions

These must be functions taking (ctx, part) and raise NotImplementedError if they cannot handle a certain part

rdflib.plugins.sparql.**SPARQL_DEFAULT_GRAPH_UNION = True**
If True - the default graph in the RDF Dataset is the union of all named graphs (like RDFLib's ConjunctiveGraph)

rdflib.plugins.sparql.**SPARQL_LOAD_GRAPHS = True**
If True, using FROM <uri> and FROM NAMED <uri> will load/parse more data

rdflib.plugins.stores package

Submodules

rdflib.plugins.stores.auditable module

This wrapper intercepts calls through the store interface and implements thread-safe logging of destructive operations (adds / removes) in reverse. This is persisted on the store instance and the reverse operations are executed In order to return the store to the state it was when the transaction began Since the reverse operations are persisted on the store, the store itself acts as a transaction.

Calls to commit or rollback, flush the list of reverse operations This provides thread-safe atomicity and isolation (assuming concurrent operations occur with different store instances), but no durability (transactions are persisted in memory and wont be available to reverse operations after the system fails): A and I out of ACID.

class rdflib.plugins.stores.auditable.**AuditableStore**(*store*)

Bases: *rdflib.store.Store*

__init__(*store*)

identifier: URIRef of the Store. Defaults to CWD configuration: string containing infomation open can use to connect to datastore.

__len__(*context=None*)

Number of statements in the store. This should only account for non- quoted (asserted) statements if the context is not specified, otherwise it should return the number of statements in the formula or context given.

Parameters **context** – a graph instance to query or None

__module__ = 'rdflib.plugins.stores.auditable'

add(*triple*, *context*, *quoted=False*)

Adds the given statement to a specific context or to the model. The quoted argument is interpreted by formula-aware stores to indicate this statement is quoted/hypothetical It should be an error to not specify a context and have the quoted argument be True. It should also be an error for the quoted argument to be True when the store is not formula-aware.

bind(*prefix*, *namespace*)

close(*commit_pending_transaction=False*)

This closes the database connection. The commit_pending_transaction parameter specifies whether to commit all pending transactions before closing (if the store is transactional).

commit()

contexts(*triple=None*)

Generator over all contexts in the graph. If triple is specified, a generator over all contexts the triple is in.

if store is graph_aware, may also return empty contexts

Returns a generator over Nodes

destroy(*configuration*)

This destroys the instance of the store identified by the configuration string.

namespace(*prefix*)

namespaces()

open(*configuration*, *create=True*)

Opens the store specified by the configuration string. If create is True a store will be created if it does not

already exist. If create is False and a store does not already exist an exception is raised. An exception is also raised if a store exists, but there is insufficient permissions to open the store. This should return one of: VALID_STORE, CORRUPTED_STORE, or NO_STORE

prefix(*namespace*)

query(**args*, ***kw*)
If stores provide their own SPARQL implementation, override this.

queryGraph is None, a URIRef or '__UNION__' If None the graph is specified in the query-string/object If URIRef it specifies the graph to query, If '__UNION__' the union of all named graphs should be queried (This is used by ConjunctiveGraphs Values other than None obviously only makes sense for context-aware stores.)

remove(*spo*, *context=None*)
Remove the set of triples matching the pattern from the store

rollback()

triples(*triple*, *context=None*)
A generator over all the triples matching the pattern. Pattern can include any objects for used for comparing against nodes in the store, for example, REGEXTerm, URIRef, Literal, BNode, Variable, Graph, QuotedGraph, Date? DateRange?

Parameters **context** – A conjunctive query can be indicated by either providing a value of None, or a specific context can be queries by passing a Graph instance (if store is context aware).

rdflib.plugins.stores.concurrent module

class rdflib.plugins.stores.concurrent.**ConcurrentStore**(*store*)
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.stores.concurrent', '__init__'

__init__(*store*)
Initialize self. See help(type(self)) for accurate signature.

__len__()

__module__ = 'rdflib.plugins.stores.concurrent'

__weakref__
list of weak references to the object (if defined)

add(*triple*)

remove(*triple*)

triples(*triple*)

class rdflib.plugins.stores.concurrent.**ResponsibleGenerator**(*gen*, *cleanup*)
Bases: object

A generator that will help clean up when it is done being used.

__del__()

__init__(*gen*, *cleanup*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()

__module__ = 'rdflib.plugins.stores.concurrent'

__next__()

__slots__ = ['cleanup', 'gen']

cleanup

gen

rdflib.plugins.stores.regexmatching module

This wrapper intercepts calls through the store interface which make use of the REGEXTerm class to represent matches by REGEX instead of literal comparison.

Implemented for stores that don't support this and essentially provides the support by replacing the REGEXTerms by wildcards (None) and matching against the results from the store it's wrapping.

class rdflib.plugins.stores.regexmatching.**REGEXMatching**(*storage*)
Bases: *rdflib.store.Store*

__init__(*storage*)
identifier: URIRef of the Store. Defaults to CWD configuration: string containing infomation open can use to connect to datastore.

__len__(*context=None*)
Number of statements in the store. This should only account for non- quoted (asserted) statements if the context is not specified, otherwise it should return the number of statements in the formula or context given.

Parameters **context** – a graph instance to query or None

__module__ = 'rdflib.plugins.stores.regexmatching'

add(*triple*, *context*, *quoted=False*)
Adds the given statement to a specific context or to the model. The quoted argument is interpreted by formula-aware stores to indicate this statement is quoted/hypothetical It should be an error to not specify a context and have the quoted argument be True. It should also be an error for the quoted argument to be True when the store is not formula-aware.

bind(*prefix*, *namespace*)

close(*commit_pending_transaction=False*)
This closes the database connection. The commit_pending_transaction parameter specifies whether to commit all pending transactions before closing (if the store is transactional).

commit()

contexts(*triple=None*)
Generator over all contexts in the graph. If triple is specified, a generator over all contexts the triple is in.

if store is graph_aware, may also return empty contexts

Returns a generator over Nodes

destroy(*configuration*)
This destroys the instance of the store identified by the configuration string.

namespace(*prefix*)

namespaces()

open(*configuration*, *create=True*)
Opens the store specified by the configuration string. If create is True a store will be created if it does not already exist. If create is False and a store does not already exist an exception is raised. An exception is also raised if a store exists, but there is insufficient permissions to open the store. This should return one of: VALID_STORE, CORRUPTED_STORE, or NO_STORE

prefix(*namespace*)

remove(*triple*, *context=None*)
Remove the set of triples matching the pattern from the store

remove_context(*identifier*)

rollback()

triples(*triple*, *context=None*)
A generator over all the triples matching the pattern. Pattern can include any objects for used for comparing against nodes in the store, for example, REGEXTerm, URIRef, Literal, BNode, Variable, Graph, QuotedGraph, Date? DateRange?

Parameters **context** – A conjunctive query can be indicated by either providing a value of None, or a specific context can be queries by passing a Graph instance (if store is context aware).

class rdflib.plugins.stores.regexmatching.**REGEXTerm**(*expr*)
Bases: str

REGEXTerm can be used in any term slot and is interpreted as a request to perform a REGEX match (not a string comparison) using the value (pre-compiled) for checking rdf:type matches

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.stores.regexmatching', '__doc_

__init__(*expr*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugins.stores.regexmatching'

__reduce__()
Helper for pickle.

__weakref__
list of weak references to the object (if defined)

rdflib.plugins.stores.regexmatching.**regexCompareQuad**(*quad*, *regexQuad*)

rdflib.plugins.stores.sparqlconnector module

class rdflib.plugins.stores.sparqlconnector.**SPARQLConnector**(*query_endpoint=None*, *update_endpoint=None*, *returnFormat='xml'*, *method='GET'*, ***kwargs*)
Bases: object

this class deals with nitty gritty details of talking to a SPARQL server

__dict__ = mappingproxy({'__module__': 'rdflib.plugins.stores.sparqlconnector', '__do

__init__(*query_endpoint=None*, *update_endpoint=None*, *returnFormat='xml'*, *method='GET'*, ***kwargs*)

Any additional keyword arguments will be passed to requests, and can be used to setup timesouts, basic auth, etc.

__module__ = 'rdflib.plugins.stores.sparqlconnector'

__weakref__

list of weak references to the object (if defined)

close()

property method

query(*query*, *default_graph=None*)

property session

update(*update*, *default_graph=None*)

exception rdflib.plugins.stores.sparqlconnector.**SPARQLConnectorException**

Bases: Exception

__module__ = 'rdflib.plugins.stores.sparqlconnector'

__weakref__

list of weak references to the object (if defined)

rdflib.plugins.stores.sparqlstore module

This is an RDFLib store around Ivan Herman et al.'s SPARQL service wrapper. This was first done in layer-cake, and then ported to RDFLib

class rdflib.plugins.stores.sparqlstore.**SPARQLStore**(*endpoint=None*, *sparql11=True*, *context_aware=True*, *node_to_sparql=<function _node_to_sparql>*, *returnFormat='xml'*, ***sparqlconnector_kwargs*)

Bases: *rdflib.plugins.stores.sparqlconnector.SPARQLConnector*, *rdflib.store.Store*

An RDFLib store around a SPARQL endpoint

This is context-aware and should work as expected when a context is specified.

For ConjunctiveGraphs, reading is done from the "default graph". Exactly what this means depends on your endpoint, because SPARQL does not offer a simple way to query the union of all graphs as it would be expected for a ConjuntiveGraph. This is why we recommend using Dataset instead, which is motivated by the SPARQL 1.1.

Fuseki/TDB has a flag for specifying that the default graph is the union of all graphs (`tdb:unionDefaultGraph` in the Fuseki config).

> **Warning:** By default the SPARQL Store does not support blank-nodes!
>
> As blank-nodes act as variables in SPARQL queries, there is no way to query for a particular blank node without using non-standard SPARQL extensions.
>
> See http://www.w3.org/TR/sparql11-query/#BGPsparqlBNodes

You can make use of such extensions through the `node_to_sparql` argument. For example if you want to transform BNode('0001') into "<bnode:b0001>", you can use a function like this:

```
>>> def my_bnode_ext(node):
...     if isinstance(node, BNode):
...         return '<bnode:b%s>' % node
...     return _node_to_sparql(node)
>>> store = SPARQLStore('http://dbpedia.org/sparql',
...                     node_to_sparql=my_bnode_ext)
```

You can request a particular result serialization with the `returnFormat` parameter. This is a string that must have a matching plugin registered. Built in is support for `xml`, `json`, `csv`, `tsv` and `application/rdf+xml`.

The underlying SPARQLConnector builds in the requests library. Any extra kwargs passed to the SPARQLStore connector are passed to requests when doing HTTP calls. I.e. you have full control of cookies/auth/headers.

Form example:

```
>>> store = SPARQLStore('...my endpoint ...', auth=('user','pass'))
```

will use HTTP basic auth.

__init__(*endpoint=None*, *sparql11=True*, *context_aware=True*, *node_to_sparql=<function _node_to_sparql>*, *returnFormat='xml'*, ***sparqlconnector_kwargs*)

__len__(*context=None*)

Number of statements in the store. This should only account for non- quoted (asserted) statements if the context is not specified, otherwise it should return the number of statements in the formula or context given.

Parameters **context** – a graph instance to query or None

__module__ = 'rdflib.plugins.stores.sparqlstore'

add(*_*, *context=None*, *quoted=False*)

Adds the given statement to a specific context or to the model. The quoted argument is interpreted by formula-aware stores to indicate this statement is quoted/hypothetical It should be an error to not specify a context and have the quoted argument be True. It should also be an error for the quoted argument to be True when the store is not formula-aware.

addN(*quads*)

Adds each item in the list of statements to a specific context. The quoted argument is interpreted by formula-aware stores to indicate this statement is quoted/hypothetical. Note that the default implementation is a redirect to add

add_graph(*graph*)

Add a graph to the store, no effect if the graph already exists. :param graph: a Graph instance

bind(*prefix*, *namespace*)

close(*commit_pending_transaction=None*)

This closes the database connection. The commit_pending_transaction parameter specifies whether to commit all pending transactions before closing (if the store is transactional).

commit()

contexts(*triple=None*)

Iterates over results to "SELECT ?NAME { GRAPH ?NAME { ?s ?p ?o } }" or "SELECT ?NAME { GRAPH ?NAME {} }" if triple is *None*.

Returns instances of this store with the SPARQL wrapper object updated via addNamedGraph(?NAME).

This causes a named-graph-uri key / value pair to be sent over the protocol.

Please note that some SPARQL endpoints are not able to find empty named graphs.

create(*configuration*)

destroy(*configuration*)
This destroys the instance of the store identified by the configuration string.

formula_aware = False

graph_aware = True

namespace(*prefix*)

namespaces()

open(*configuration*, *create=False*)
sets the endpoint URL for this SPARQLStore if create==True an exception is thrown.

prefix(*namespace*)

query(*query*, *initNs={}*, *initBindings={}*, *queryGraph=None*, *DEBUG=False*)
If stores provide their own SPARQL implementation, override this.

queryGraph is None, a URIRef or '__UNION__' If None the graph is specified in the query-string/object If URIRef it specifies the graph to query, If '__UNION__' the union of all named graphs should be queried (This is used by ConjunctiveGraphs Values other than None obviously only makes sense for context-aware stores.)

regex_matching = 0

remove(*_*, *context*)
Remove the set of triples matching the pattern from the store

remove_graph(*graph*)
Remove a graph from the store, this shoud also remove all triples in the graph

Parameters **graphid** – a Graph instance

rollback()

transaction_aware = False

triples(*spo*, *context=None*)

- **tuple (s, o, p)** the triple used as filter for the SPARQL select. (None, None, None) means anything.
- **context context** the graph effectively calling this method.

Returns a tuple of triples executing essentially a SPARQL like SELECT ?subj ?pred ?obj WHERE { ?subj ?pred ?obj }

context may include three parameter to refine the underlying query:

- LIMIT: an integer to limit the number of results
- OFFSET: an integer to enable paging of results
- ORDERBY: an instance of Variable('s'), Variable('o') or Variable('p')

or, by default, the first 'None' from the given triple

- Using LIMIT or OFFSET automatically include ORDERBY otherwise this is

because the results are retrieved in a not deterministic way (depends on the walking path on the graph) - Using OFFSET without defining LIMIT will discard the first OFFSET - 1 results

`` a_graph.LIMIT = limit a_graph.OFFSET = offset triple_generator = a_graph.triples(mytriple):

#do something

#Removes LIMIT and OFFSET if not required for the next triple() calls del a_graph.LIMIT del a_graph.OFFSET ``

triples_choices(*_*, *context=None*)

A variant of triples that can take a list of terms instead of a single term in any slot. Stores can implement this to optimize the response time from the import default 'fallback' implementation, which will iterate over each term in the list and dispatch to triples.

class rdflib.plugins.stores.sparqlstore.**SPARQLUpdateStore**(*queryEndpoint=None*, *update_endpoint=None*, *sparql11=True*, *context_aware=True*, *postAsEncoded=True*, *autocommit=True*, *dirty_reads=False*, ***kwds*)

Bases: *rdflib.plugins.stores.sparqlstore.SPARQLStore*

A store using SPARQL queries for reading and SPARQL Update for changes.

This can be context-aware, if so, any changes will be to the given named graph only.

In favor of the SPARQL 1.1 motivated Dataset, we advise against using this with ConjunctiveGraphs, as it reads and writes from and to the "default graph". Exactly what this means depends on the endpoint and can result in confusion.

For Graph objects, everything works as expected.

See the *SPARQLStore* base class for more information.

BLOCK_END = '}'

BLOCK_FINDING_PATTERN = re.compile('(?P<block_start>{)|(?P<block_end>})|(?P<block_cont

BLOCK_START = '{'

BlockContent = '((\'([^\'\\\\]|\\\\.)*\')|("([^"\\\\]|\\\\.)*")|(\'\'\'((\'|\'\')?([^

BlockFinding = '(?P<block_start>{)|(?P<block_end>})|(?P<block_content>((\'([^\'\\\\]|

COMMENT = '#[^\\x0D\\x0A]*([\\x0D\\x0A]|\\Z)'

ESCAPED = '\\\\.'

IRIREF = '<([^<>"{}|^`\\]\\\\\\[\\x00-\\x20])*>'

STRING_LITERAL1 = "'([^'\\\\]|\\\\.)*'"

STRING_LITERAL2 = '"([^"\\\\]|\\\\.)*"'

STRING_LITERAL_LONG1 = "'''(('|'')?([^'\\\\]|\\\\.))*'''"

STRING_LITERAL_LONG2 = '"""(("|"")?([^"\\\\]|\\\\.))*"""'

String = '(\'([^\'\\\\]|\\\\.)*\')|("([^"\\\\]|\\\\.)*")|(\'\'\'((\'|\'\')?([^\'\\\\]|

__init__(*queryEndpoint=None*, *update_endpoint=None*, *sparql11=True*, *context_aware=True*, *postAsEncoded=True*, *autocommit=True*, *dirty_reads=False*, ***kwds*)

:param autocommit if set, the store will commit after every writing operations. If False, we only make queries on the server once commit is called.

:param dirty_reads if set, we do not commit before reading. So you cannot read what you wrote before manually calling commit.

__len__(**args*, ***kwargs*)
Number of statements in the store. This should only account for non- quoted (asserted) statements if the context is not specified, otherwise it should return the number of statements in the formula or context given.

Parameters **context** – a graph instance to query or None

__module__ = 'rdflib.plugins.stores.sparqlstore'

add(*spo*, *context=None*, *quoted=False*)
Add a triple to the store of triples.

addN(*quads*)
Add a list of quads to the store.

add_graph(*graph*)
Add a graph to the store, no effect if the graph already exists. :param graph: a Graph instance

close(*commit_pending_transaction=False*)
This closes the database connection. The commit_pending_transaction parameter specifies whether to commit all pending transactions before closing (if the store is transactional).

commit()
add(), addN(), and remove() are transactional to reduce overhead of many small edits. Read and update() calls will automatically commit any outstanding edits. This should behave as expected most of the time, except that alternating writes and reads can degenerate to the original call-per-triple situation that originally existed.

contexts(**args*, ***kwargs*)
Iterates over results to "SELECT ?NAME { GRAPH ?NAME { ?s ?p ?o } }" or "SELECT ?NAME { GRAPH ?NAME {} }" if triple is *None*.

Returns instances of this store with the SPARQL wrapper object updated via addNamedGraph(?NAME).

This causes a named-graph-uri key / value pair to be sent over the protocol.

Please note that some SPARQL endpoints are not able to find empty named graphs.

open(*configuration*, *create=False*)
sets the endpoint URLs for this SPARQLStore :param configuration: either a tuple of (query_endpoint, update_endpoint),

or a string with the endpoint which is configured as query and update endpoint

Parameters **create** – if True an exception is thrown.

query(**args*, ***kwargs*)
If stores provide their own SPARQL implementation, override this.

queryGraph is None, a URIRef or '__UNION__' If None the graph is specified in the query-string/object If URIRef it specifies the graph to query, If '__UNION__' the union of all named graphs should be queried (This is used by ConjunctiveGraphs Values other than None obviously only makes sense for context-aware stores.)

remove(*spo*, *context*)
Remove a triple from the store

remove_graph(*graph*)
Remove a graph from the store, this shoud also remove all triples in the graph

Parameters **graphid** – a Graph instance

rollback()

setTimeout(*timeout*)

triples(**args*, ***kwargs*)

- **tuple (s, o, p)** the triple used as filter for the SPARQL select. (None, None, None) means anything.
- **context context** the graph effectively calling this method.

Returns a tuple of triples executing essentially a SPARQL like SELECT ?subj ?pred ?obj WHERE { ?subj ?pred ?obj }

context may include three parameter to refine the underlying query:

- LIMIT: an integer to limit the number of results
- OFFSET: an integer to enable paging of results
- ORDERBY: an instance of Variable('s'), Variable('o') or Variable('p')

or, by default, the first 'None' from the given triple

- Using LIMIT or OFFSET automatically include ORDERBY otherwise this is

because the results are retrieved in a not deterministic way (depends on the walking path on the graph) - Using OFFSET without defining LIMIT will discard the first OFFSET - 1 results

`` a_graph.LIMIT = limit a_graph.OFFSET = offset triple_generator = a_graph.triples(mytriple):

#do something

#Removes LIMIT and OFFSET if not required for the next triple() calls del a_graph.LIMIT del a_graph.OFFSET ``

update(*query*, *initNs={}*, *initBindings={}*, *queryGraph=None*, *DEBUG=False*)

Perform a SPARQL Update Query against the endpoint, INSERT, LOAD, DELETE etc. Setting initNs adds PREFIX declarations to the beginning of the update. Setting initBindings adds inline VALUEs to the beginning of every WHERE clause. By the SPARQL grammar, all operations that support variables (namely INSERT and DELETE) require a WHERE clause. Important: initBindings fails if the update contains the substring 'WHERE {' which does not denote a WHERE clause, e.g. if it is part of a literal.

Context-aware query rewriting

- **When:** If context-awareness is enabled and the graph is not the default graph of the store.
- **Why:** To ensure consistency with the *IOMemory* store. The graph must except "local" SPARQL requests (requests with no GRAPH keyword) like if it was the default graph.
- **What is done:** These "local" queries are rewritten by this store. The content of each block of a SPARQL Update operation is wrapped in a GRAPH block except if the block is empty. This basically causes INSERT, INSERT DATA, DELETE, DELETE DATA and WHERE to operate only on the context.
- **Example:** *"INSERT DATA { <urn:michel> <urn:likes> <urn:pizza> }"* is converted into *"INSERT DATA { GRAPH <urn:graph> { <urn:michel> <urn:likes> <urn:pizza> } }"*.
- **Warning:** Queries are presumed to be "local" but this assumption is **not checked**. For instance, if the query already contains GRAPH blocks, the latter will be wrapped in new GRAPH blocks.
- **Warning:** A simplified grammar is used that should tolerate extensions of the SPARQL grammar. Still, the process may fail in uncommon situations and produce invalid output.

```
where_pattern = re.compile('(?P<where>WHERE\\s*\\{)', re.IGNORECASE)
```

Module contents

This package contains modules for additional RDFLib stores

Submodules

rdflib.plugins.memory module

class rdflib.plugins.memory.**Memory**(*configuration=None*, *identifier=None*)
Bases: *rdflib.store.Store*

An in memory implementation of a triple store.

This triple store uses nested dictionaries to store triples. Each triple is stored in two such indices as follows spo[s][p][o] = 1 and pos[p][o][s] = 1.

Authors: Michel Pelletier, Daniel Krech, Stefan Niederhauser

__init__(*configuration=None*, *identifier=None*)
identifier: URIRef of the Store. Defaults to CWD configuration: string containing infomation open can use to connect to datastore.

__len__(*context=None*)
Number of statements in the store. This should only account for non- quoted (asserted) statements if the context is not specified, otherwise it should return the number of statements in the formula or context given.

Parameters **context** – a graph instance to query or None

__module__ = 'rdflib.plugins.memory'

add(*triple*, *context*, *quoted=False*)
Add a triple to the store of triples.

bind(*prefix*, *namespace*)

namespace(*prefix*)

namespaces()

prefix(*namespace*)

remove(*triple_pattern*, *context=None*)
Remove the set of triples matching the pattern from the store

triples(*triple_pattern*, *context=None*)
A generator over all the triples matching

class rdflib.plugins.memory.**IOMemory**(*configuration=None*, *identifier=None*)
Bases: *rdflib.store.Store*

An integer-key-optimized context-aware in-memory store.

Uses three dict indices (for subjects, objects and predicates) holding sets of triples. Context information is tracked in a separate dict, with the triple as key and a dict of {context: quoted} items as value. The context information is used to filter triple query results.

Memory usage is low due to several optimizations. RDF nodes are not stored directly in the indices; instead, the indices hold integer keys and the actual nodes are only stored once in int-to-object and object-to-int mapping

dictionaries. A default context is determined based on the first triple that is added to the store, and no context information is actually stored for subsequent other triples with the same context information.

Most operations should be quite fast, but a triples() query with two bound parts requires a set intersection operation, which may be slow in some cases. When multiple contexts are used in the same store, filtering based on context has to be done after each query, which may also be slow.

__init__(*configuration=None*, *identifier=None*)
identifier: URIRef of the Store. Defaults to CWD configuration: string containing infomation open can use to connect to datastore.

__len__(*context=None*)
Number of statements in the store. This should only account for non- quoted (asserted) statements if the context is not specified, otherwise it should return the number of statements in the formula or context given.

Parameters **context** – a graph instance to query or None

__module__ = 'rdflib.plugins.memory'

add(*triple*, *context*, *quoted=False*)
Adds the given statement to a specific context or to the model. The quoted argument is interpreted by formula-aware stores to indicate this statement is quoted/hypothetical It should be an error to not specify a context and have the quoted argument be True. It should also be an error for the quoted argument to be True when the store is not formula-aware.

add_graph(*graph*)
Add a graph to the store, no effect if the graph already exists. :param graph: a Graph instance

bind(*prefix*, *namespace*)

context_aware = True

contexts(*triple=None*)
Generator over all contexts in the graph. If triple is specified, a generator over all contexts the triple is in.

if store is graph_aware, may also return empty contexts

Returns a generator over Nodes

formula_aware = True

graph_aware = True

namespace(*prefix*)

namespaces()

prefix(*namespace*)

remove(*triplepat*, *context=None*)
Remove the set of triples matching the pattern from the store

remove_graph(*graph*)
Remove a graph from the store, this shoud also remove all triples in the graph

Parameters **graphid** – a Graph instance

triples(*triplein*, *context=None*)
A generator over all the triples matching the pattern. Pattern can include any objects for used for comparing against nodes in the store, for example, REGEXTerm, URIRef, Literal, BNode, Variable, Graph, QuotedGraph, Date? DateRange?

Parameters **context** – A conjunctive query can be indicated by either providing a value of None, or a specific context can be queries by passing a Graph instance (if store is context aware).

rdflib.plugins.sleepycat module

class rdflib.plugins.sleepycat.**Sleepycat**(*configuration=None*, *identifier=None*)

Bases: *rdflib.store.Store*

__init__(*configuration=None*, *identifier=None*)

identifier: URIRef of the Store. Defaults to CWD configuration: string containing infomation open can use to connect to datastore.

__len__(*context=None*)

Number of statements in the store. This should only account for non- quoted (asserted) statements if the context is not specified, otherwise it should return the number of statements in the formula or context given.

Parameters **context** – a graph instance to query or None

__module__ = 'rdflib.plugins.sleepycat'

add(*triple*, *context*, *quoted=False*, *txn=None*)

Add a triple to the store of triples.

add_graph(*graph*)

Add a graph to the store, no effect if the graph already exists. :param graph: a Graph instance

bind(*prefix*, *namespace*)

close(*commit_pending_transaction=False*)

This closes the database connection. The commit_pending_transaction parameter specifies whether to commit all pending transactions before closing (if the store is transactional).

context_aware = True

contexts(*triple=None*)

Generator over all contexts in the graph. If triple is specified, a generator over all contexts the triple is in.

if store is graph_aware, may also return empty contexts

Returns a generator over Nodes

db_env = None

formula_aware = True

graph_aware = True

property identifier

is_open()

namespace(*prefix*)

namespaces()

open(*path*, *create=True*)

Opens the store specified by the configuration string. If create is True a store will be created if it does not already exist. If create is False and a store does not already exist an exception is raised. An exception is also raised if a store exists, but there is insufficient permissions to open the store. This should return one of: VALID_STORE, CORRUPTED_STORE, or NO_STORE

prefix(*namespace*)

remove(*spo*, *context*, *txn=None*)
Remove the set of triples matching the pattern from the store

remove_graph(*graph*)
Remove a graph from the store, this shoud also remove all triples in the graph

Parameters **graphid** – a Graph instance

sync()

transaction_aware = False

triples(*spo*, *context=None*, *txn=None*)
A generator over all the triples matching

Module contents

Default plugins for rdflib.

This is a namespace package and contains the default plugins for rdflib.

rdflib.tools package

Submodules

rdflib.tools.csv2rdf module

A commandline tool for semi-automatically converting CSV to RDF

try: `csv2rdf --help`

class rdflib.tools.csv2rdf.**CSV2RDF**
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.tools.csv2rdf', '__init__': <function

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.tools.csv2rdf'

__weakref__
list of weak references to the object (if defined)

convert(*csvreader*)

triple(*s*, *p*, *o*)

rdflib.tools.graphisomorphism module

A commandline tool for testing if RDF graphs are isomorpic, i.e. equal if BNode labels are ignored.

class rdflib.tools.graphisomorphism.**IsomorphicTestableGraph**(***kargs*)
Bases: *rdflib.graph.Graph*

Ported from: http://www.w3.org/2001/sw/DataAccess/proto-tests/tools/rdfdiff.py (Sean B Palmer's RDF Graph Isomorphism Tester)

__eq__(*G*)
Graph isomorphism testing.

__hash__ = None

__init__(***kargs*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.tools.graphisomorphism'

__ne__(*G*)
Negative graph isomorphism testing.

hashtriples()

internal_hash()
This is defined instead of __hash__ to avoid a circular recursion scenario with the Memory store for rdflib which requires a hash lookup in order to return a generator of triples

vhash(*term*, *done=False*)

vhashtriple(*triple*, *term*, *done*)

vhashtriples(*term*, *done*)

rdflib.tools.graphisomorphism.**main**()

rdflib.tools.rdf2dot module

A commandline tool for drawing RDF graphs in Graphviz DOT format

You can draw the graph of an RDF file directly:

rdflib.tools.rdf2dot.**main**()

rdflib.tools.rdf2dot.**rdf2dot**(*g*, *stream*, *opts={}*)
Convert the RDF graph to DOT writes the dot output to the stream

rdflib.tools.rdfpipe module

A commandline tool for parsing RDF in different formats and serializing the resulting graph to a chosen format.

rdflib.tools.rdfpipe.**main**()

rdflib.tools.rdfpipe.**make_option_parser**()

rdflib.tools.rdfpipe.**parse_and_serialize**(*input_files*, *input_format*, *guess*, *outfile*, *output_format*, *ns_bindings*, *store_conn=''*, *store_type=None*)

rdflib.tools.rdfs2dot module

A commandline tool for drawing RDFS Class diagrams in Graphviz DOT format

You can draw the graph of an RDFS file directly:

rdflib.tools.rdfs2dot.**main**()

rdflib.tools.rdfs2dot.**rdfs2dot**(*g*, *stream*, *opts={}*)
Convert the RDFS schema in a graph writes the dot output to the stream

Module contents

Various commandline tools for working with RDFLib

Submodules

rdflib.collection module

class rdflib.collection.**Collection**(*graph*, *uri*, *seq=[]*)
Bases: object

See "Emulating container types": https://docs.python.org/reference/datamodel.html#emulating-container-types

```
>>> from rdflib.graph import Graph
>>> from pprint import pprint
>>> listName = BNode()
>>> g = Graph('IOMemory')
>>> listItem1 = BNode()
>>> listItem2 = BNode()
>>> g.add((listName, RDF.first, Literal(1)))
>>> g.add((listName, RDF.rest, listItem1))
>>> g.add((listItem1, RDF.first, Literal(2)))
>>> g.add((listItem1, RDF.rest, listItem2))
>>> g.add((listItem2, RDF.rest, RDF.nil))
>>> g.add((listItem2, RDF.first, Literal(3)))
>>> c = Collection(g,listName)
>>> pprint([term.n3() for term in c])
[u'"1"^^<http://www.w3.org/2001/XMLSchema#integer>',
 u'"2"^^<http://www.w3.org/2001/XMLSchema#integer>',
 u'"3"^^<http://www.w3.org/2001/XMLSchema#integer>']
```

```
>>> Literal(1) in c
True
>>> len(c)
3
>>> c._get_container(1) == listItem1
True
>>> c.index(Literal(2)) == 1
True
```

__delitem__(*key*)

```
>>> from rdflib.namespace import RDF, RDFS
>>> from rdflib import Graph
>>> from pprint import pformat
>>> g = Graph()
>>> a = BNode('foo')
>>> b = BNode('bar')
>>> c = BNode('baz')
>>> g.add((a, RDF.first, RDF.type))
>>> g.add((a, RDF.rest, b))
>>> g.add((b, RDF.first, RDFS.label))
>>> g.add((b, RDF.rest, c))
>>> g.add((c, RDF.first, RDFS.comment))
>>> g.add((c, RDF.rest, RDF.nil))
>>> len(g)
6
>>> def listAncestry(node, graph):
...   for i in graph.subjects(RDF.rest, node):
...     yield i
>>> [str(node.n3())
...   for node in g.transitiveClosure(listAncestry, RDF.nil)]
['_:baz', '_:bar', '_:foo']
>>> lst = Collection(g, a)
>>> len(lst)
3
>>> b == lst._get_container(1)
True
>>> c == lst._get_container(2)
True
>>> del lst[1]
>>> len(lst)
2
>>> len(g)
4
```

__dict__ = mappingproxy({'__module__': 'rdflib.collection', '__doc__': '\n See "Emul

__getitem__(*key*)
TODO

__iadd__(*other*)

__init__(*graph*, *uri*, *seq=[]*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()
Iterator over items in Collections

__len__()
length of items in collection.

__module__ = 'rdflib.collection'

__setitem__(*key*, *value*)
TODO

__weakref__
list of weak references to the object (if defined)

append(*item*)

```
>>> from rdflib.graph import Graph
>>> listName = BNode()
>>> g = Graph()
>>> c = Collection(g,listName,[Literal(1),Literal(2)])
>>> links = [
...     list(g.subjects(object=i, predicate=RDF.first))[0] for i in c]
>>> len([i for i in links if (i, RDF.rest, RDF.nil) in g])
1
```

clear()

index(*item*)
Returns the 0-based numerical index of the item in the list

n3()

```
>>> from rdflib.graph import Graph
>>> listName = BNode()
>>> g = Graph('IOMemory')
>>> listItem1 = BNode()
>>> listItem2 = BNode()
>>> g.add((listName, RDF.first, Literal(1)))
>>> g.add((listName, RDF.rest, listItem1))
>>> g.add((listItem1, RDF.first, Literal(2)))
>>> g.add((listItem1, RDF.rest, listItem2))
>>> g.add((listItem2, RDF.rest, RDF.nil))
>>> g.add((listItem2, RDF.first, Literal(3)))
>>> c = Collection(g, listName)
>>> print(c.n3())
( "1"^^<http://www.w3.org/2001/XMLSchema#integer>
  "2"^^<http://www.w3.org/2001/XMLSchema#integer>
  "3"^^<http://www.w3.org/2001/XMLSchema#integer> )
```

rdflib.compare module

A collection of utilities for canonicalizing and inspecting graphs.

Among other things, they solve of the problem of deterministic bnode comparisons.

Warning: the time to canonicalize bnodes may increase exponentially on degenerate larger graphs. Use with care!

Example of comparing two graphs:

```
>>> g1 = Graph().parse(format='n3', data='''
...     @prefix : <http://example.org/ns#> .
...     <http://example.org> :rel
...         <http://example.org/same>,
...         [ :label "Same" ],
...         <http://example.org/a>,
...         [ :label "A" ] .
... ''')
>>> g2 = Graph().parse(format='n3', data='''
...     @prefix : <http://example.org/ns#> .
...     <http://example.org> :rel
...         <http://example.org/same>,
...         [ :label "Same" ],
```

(continues on next page)

(continued from previous page)

```
...          <http://example.org/b>,
...          [ :label "B" ] .
... ''')
>>>
>>> iso1 = to_isomorphic(g1)
>>> iso2 = to_isomorphic(g2)
```

These are not isomorphic:

```
>>> iso1 == iso2
False
```

Diff the two graphs:

```
>>> in_both, in_first, in_second = graph_diff(iso1, iso2)
```

Present in both:

```
>>> def dump_nt_sorted(g):
...     for l in sorted(g.serialize(format='nt').splitlines()):
...         if l: print(l.decode('ascii'))

>>> dump_nt_sorted(in_both)
<http://example.org>
    <http://example.org/ns#rel> <http://example.org/same> .
<http://example.org>
    <http://example.org/ns#rel> _:cbcaabaaba17fecbc304a64f8edee4335e .
_:cbcaabaaba17fecbc304a64f8edee4335e
    <http://example.org/ns#label> "Same" .
```

Only in first:

```
>>> dump_nt_sorted(in_first)
<http://example.org>
    <http://example.org/ns#rel> <http://example.org/a> .
<http://example.org>
    <http://example.org/ns#rel> _:cb124e4c6da0579f810c0ffe4eff485bd9 .
_:cb124e4c6da0579f810c0ffe4eff485bd9
    <http://example.org/ns#label> "A" .
```

Only in second:

```
>>> dump_nt_sorted(in_second)
<http://example.org>
    <http://example.org/ns#rel> <http://example.org/b> .
<http://example.org>
    <http://example.org/ns#rel> _:cb558f30e21ddfc05ca53108348338ade8 .
_:cb558f30e21ddfc05ca53108348338ade8
    <http://example.org/ns#label> "B" .
```

class rdflib.compare.**IsomorphicGraph**(***kwargs*)

Bases: *rdflib.graph.ConjunctiveGraph*

An implementation of the RGDA1 graph digest algorithm.

An implementation of RGDA1 (publication below), a combination of Sayers & Karp's graph digest algorithm using sum and SHA-256 <http://www.hpl.hp.com/techreports/2003/HPL-2003-235R1.pdf> and traces <http://pallini.di.uniroma1.it>, an average case polynomial time algorithm for graph canonicalization.

McCusker, J. P. (2015). WebSig: A Digital Signature Framework for the Web. Rensselaer Polytechnic Institute, Troy, NY. http://gradworks.umi.com/3727015.pdf

__eq__(*other*)
Graph isomorphism testing.

__hash__()
Return hash(self).

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.compare'

__ne__(*other*)
Negative graph isomorphism testing.

graph_digest(*stats=None*)
Synonym for IsomorphicGraph.internal_hash.

internal_hash(*stats=None*)
This is defined instead of __hash__ to avoid a circular recursion scenario with the Memory store for rdflib which requires a hash lookup in order to return a generator of triples.

rdflib.compare.**to_isomorphic**(*graph*)

rdflib.compare.**isomorphic**(*graph1*, *graph2*)
Compare graph for equality.

Uses an algorithm to compute unique hashes which takes bnodes into account.

Examples:

```
>>> g1 = Graph().parse(format='n3', data='''
...     @prefix : <http://example.org/ns#> .
...     <http://example.org> :rel <http://example.org/a> .
...     <http://example.org> :rel <http://example.org/b> .
...     <http://example.org> :rel [ :label "A bnode." ] .
... ''')
>>> g2 = Graph().parse(format='n3', data='''
...     @prefix ns: <http://example.org/ns#> .
...     <http://example.org> ns:rel [ ns:label "A bnode." ] .
...     <http://example.org> ns:rel <http://example.org/b>,
...             <http://example.org/a> .
... ''')
>>> isomorphic(g1, g2)
True

>>> g3 = Graph().parse(format='n3', data='''
...     @prefix : <http://example.org/ns#> .
...     <http://example.org> :rel <http://example.org/a> .
...     <http://example.org> :rel <http://example.org/b> .
...     <http://example.org> :rel <http://example.org/c> .
... ''')
>>> isomorphic(g1, g3)
False
```

rdflib.compare.**to_canonical_graph**(*g1*, *stats=None*)
Creates a canonical, read-only graph.

Creates a canonical, read-only graph where all bnode id:s are based on deterministical SHA-256 checksums, correlated with the graph contents.

rdflib.compare.**graph_diff**(*g1*, *g2*)
Returns three sets of triples: "in both", "in first" and "in second".

rdflib.compare.**similar**(*g1*, *g2*)
Checks if the two graphs are "similar".

Checks if the two graphs are "similar", by comparing sorted triples where all bnodes have been replaced by a singular mock bnode (the _MOCK_BNODE).

This is a much cheaper, but less reliable, alternative to the comparison algorithm in isomorphic.

rdflib.compat module

Utility functions and objects to ease Python 2/3 compatibility, and different versions of support libraries.

rdflib.compat.**ascii**(*stream*)

rdflib.compat.**bopen**(**args*, ***kwargs*)

rdflib.compat.**cast_bytes**(*s*, *enc='utf-8'*)

rdflib.compat.**decodeStringEscape**(*s*)
s is byte-string - replace escapes in string

rdflib.compat.**decodeUnicodeEscape**(*s*)
s is a unicode string replace \n and \u00AC unicode escapes

rdflib.compat.**sign**(*n*)

rdflib.container module

class rdflib.container.**Container**(*graph*, *uri*, *seq=[]*, *rtype='Bag'*)
Bases: object

A class for constructing RDF containers, as per https://www.w3.org/TR/rdf11-mt/#rdf-containers

Basic usage, creating a Bag and adding to it:

```
>>> from rdflib import Graph, BNode, Literal, Bag
>>> g = Graph()
>>> b = Bag(g, BNode(), [Literal("One"), Literal("Two"), Literal("Three")])
>>> print(g.serialize(format="turtle").decode())
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

[] a rdf:Bag ;
    rdf:_1 "One" ;
    rdf:_2 "Two" ;
    rdf:_3 "Three" .



>>> # print out an item using an index reference
>>> print(b[2])
Two

>>> # add a new item
>>> b.append(Literal("Hello"))
>>> print(g.serialize(format="turtle").decode())
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
```

(continues on next page)

(continued from previous page)

```
[] a rdf:Bag ;
    rdf:_1 "One" ;
    rdf:_2 "Two" ;
    rdf:_3 "Three" ;
    rdf:_4 "Hello" .
```

__delitem__(*key*)
Removing the item with index key or predicate rdf:_key

__dict__ = mappingproxy({'__module__': 'rdflib.container', '__doc__': 'A class for c

__getitem__(*key*)
Returns item of the container at index key

__init__(*graph*, *uri*, *seq=[]*, *rtype='Bag'*)
Creates a Container

Parameters

- **graph** – a Graph instance
- **uri** – URI or Blank Node of the Container
- **seq** – the elements of the Container
- **rtype** – the type of Container, one of “Bag”, “Seq” or “Alt”

__len__()
Number of items in container

__module__ = 'rdflib.container'

__setitem__(*key*, *value*)
Sets the item at index key or predicate rdf:_key of the container to value

__weakref__
list of weak references to the object (if defined)

append(*item*)
Adding item to the end of the container

append_multiple(*other*)
Adding multiple elements to the container to the end which are in python list other

clear()
Removing all elements from the container

end()

index(*item*)
Returns the 1-based numerical index of the item in the container

items()
Returns a list of all items in the container

n3()

type_of_conatiner()

class rdflib.container.**Bag**(*graph*, *uri*, *seq=[]*)
Bases: *rdflib.container.Container*

Unordered container (no preference order of elements)

__init__(*graph*, *uri*, *seq=[]*)
Creates a Container

Parameters

- **graph** – a Graph instance
- **uri** – URI or Blank Node of the Container
- **seq** – the elements of the Container
- **rtype** – the type of Container, one of "Bag", "Seq" or "Alt"

__module__ = 'rdflib.container'

class rdflib.container.**Seq**(*graph*, *uri*, *seq=[]*)
Bases: *rdflib.container.Container*

__init__(*graph*, *uri*, *seq=[]*)
Creates a Container

Parameters

- **graph** – a Graph instance
- **uri** – URI or Blank Node of the Container
- **seq** – the elements of the Container
- **rtype** – the type of Container, one of "Bag", "Seq" or "Alt"

__module__ = 'rdflib.container'

add_at_position(*pos*, *item*)

class rdflib.container.**Alt**(*graph*, *uri*, *seq=[]*)
Bases: *rdflib.container.Container*

__init__(*graph*, *uri*, *seq=[]*)
Creates a Container

Parameters

- **graph** – a Graph instance
- **uri** – URI or Blank Node of the Container
- **seq** – the elements of the Container
- **rtype** – the type of Container, one of "Bag", "Seq" or "Alt"

__module__ = 'rdflib.container'

anyone()

exception rdflib.container.**NoElementException**(*message='rdf:Alt Container is empty'*)
Bases: Exception

__init__(*message='rdf:Alt Container is empty'*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.container'

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

rdflib.events module

Dirt Simple Events

A Dispatcher (or a subclass of Dispatcher) stores event handlers that are 'fired' simple event objects when interesting things happen.

Create a dispatcher:

```
>>> d = Dispatcher()
```

Now create a handler for the event and subscribe it to the dispatcher to handle Event events. A handler is a simple function or method that accepts the event as an argument:

```
>>> def handler1(event): print(repr(event))
>>> d.subscribe(Event, handler1)
```

Now dispatch a new event into the dispatcher, and see handler1 get fired:

```
>>> d.dispatch(Event(foo='bar', data='yours', used_by='the event handlers'))
<rdflib.events.Event ['data', 'foo', 'used_by']>
```

class rdflib.events.**Event**(***kw*)
Bases: object

An event is a container for attributes. The source of an event creates this object, or a subclass, gives it any kind of data that the events handlers need to handle the event, and then calls notify(event).

The target of an event registers a function to handle the event it is interested with subscribe(). When a sources calls notify(event), each subscriber to that event will be called in no particular order.

__dict__ = mappingproxy({'__module__': 'rdflib.events', '__doc__': '\n An event is a

__init__(***kw*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.events'

__repr__()
Return repr(self).

__weakref__
list of weak references to the object (if defined)

class rdflib.events.**Dispatcher**
Bases: object

An object that can dispatch events to a privately managed group of subscribers.

__dict__ = mappingproxy({'__module__': 'rdflib.events', '__doc__': '\n An object tha

__module__ = 'rdflib.events'

__weakref__
list of weak references to the object (if defined)

dispatch(*event*)
Dispatch the given event to the subscribed handlers for the event's type

get_map()

set_map(*amap*)

subscribe(*event_type*, *handler*)
Subscribe the given handler to an event_type. Handlers are called in the order they are subscribed.

rdflib.exceptions module

TODO:

exception rdflib.exceptions.**Error**(*msg=None*)
Bases: Exception

Base class for rdflib exceptions.

__init__(*msg=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.exceptions'

__weakref__
list of weak references to the object (if defined)

exception rdflib.exceptions.**TypeCheckError**(*node*)
Bases: *rdflib.exceptions.Error*

Parts of assertions are subject to type checks.

__init__(*node*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.exceptions'

exception rdflib.exceptions.**SubjectTypeError**(*node*)
Bases: *rdflib.exceptions.TypeCheckError*

Subject of an assertion must be an instance of URIRef.

__init__(*node*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.exceptions'

exception rdflib.exceptions.**PredicateTypeError**(*node*)
Bases: *rdflib.exceptions.TypeCheckError*

Predicate of an assertion must be an instance of URIRef.

__init__(*node*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.exceptions'

exception rdflib.exceptions.**ObjectTypeError**(*node*)
Bases: *rdflib.exceptions.TypeCheckError*

Object of an assertion must be an instance of URIRef, Literal, or BNode.

__init__(*node*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.exceptions'

exception rdflib.exceptions.**ContextTypeError**(*node*)
Bases: *rdflib.exceptions.TypeCheckError*

Context of an assertion must be an instance of URIRef.

__init__(*node*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.exceptions'

exception rdflib.exceptions.**ParserError**(*msg*)
Bases: *rdflib.exceptions.Error*

RDF Parser error.

__init__(*msg*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.exceptions'

__str__()
Return str(self).

rdflib.graph module

RDFLib defines the following kinds of Graphs:

- *Graph*
- *QuotedGraph*
- *ConjunctiveGraph*
- *Dataset*

Graph

An RDF graph is a set of RDF triples. Graphs support the python `in` operator, as well as iteration and some operations like union, difference and intersection.

see *Graph*

Conjunctive Graph

A Conjunctive Graph is the most relevant collection of graphs that are considered to be the boundary for closed world assumptions. This boundary is equivalent to that of the store instance (which is itself uniquely identified and distinct from other instances of `Store` that signify other Conjunctive Graphs). It is equivalent to all the named graphs within it and associated with a `_default_` graph which is automatically assigned a `BNode` for an identifier - if one isn't given.

see *ConjunctiveGraph*

Quoted graph

The notion of an RDF graph [14] is extended to include the concept of a formula node. A formula node may occur wherever any other kind of node can appear. Associated with a formula node is an RDF graph that is completely disjoint from all other graphs; i.e. has no nodes in common with any other graph. (It may contain the same labels as other RDF graphs; because this is, by definition, a separate graph, considerations of tidiness do not apply between the graph at a formula node and any other graph.)

This is intended to map the idea of "{ N3-expression }" that is used by N3 into an RDF graph upon which RDF semantics is defined.

see *QuotedGraph*

Dataset

The RDF 1.1 Dataset, a small extension to the Conjunctive Graph. The primary term is "graphs in the datasets" and not "contexts with quads" so there is a separate method to set/retrieve a graph in a dataset and to operate with dataset graphs. As a consequence of this approach, dataset graphs cannot be identified with blank nodes, a name is always required (RDFLib will automatically add a name if one is not provided at creation time). This implementation includes a convenience method to directly add a single quad to a dataset graph.

see *Dataset*

Working with graphs

Instantiating Graphs with default store (IOMemory) and default identifier (a BNode):

```
>>> g = Graph()
>>> g.store.__class__
<class 'rdflib.plugins.memory.IOMemory'>
>>> g.identifier.__class__
<class 'rdflib.term.BNode'>
```

Instantiating Graphs with a IOMemory store and an identifier - <http://rdflib.net>:

```
>>> g = Graph('IOMemory', URIRef("http://rdflib.net"))
>>> g.identifier
rdflib.term.URIRef('http://rdflib.net')
>>> str(g)
"<http://rdflib.net> a rdfg:Graph;rdflib:storage
 [a rdflib:Store;rdfs:label 'IOMemory']."
```

Creating a ConjunctiveGraph - The top level container for all named Graphs in a "database":

```
>>> g = ConjunctiveGraph()
>>> str(g.default_context)
"[a rdfg:Graph;rdflib:storage [a rdflib:Store;rdfs:label 'IOMemory']]."
```

Adding / removing reified triples to Graph and iterating over it directly or via triple pattern:

```
>>> g = Graph()
>>> statementId = BNode()
>>> print(len(g))
0
>>> g.add((statementId, RDF.type, RDF.Statement))
```

(continues on next page)

(continued from previous page)

```
>>> g.add((statementId, RDF.subject,
...     URIRef("http://rdflib.net/store/ConjunctiveGraph")))
>>> g.add((statementId, RDF.predicate, RDFS.label))
>>> g.add((statementId, RDF.object, Literal("Conjunctive Graph")))
>>> print(len(g))
4
>>> for s, p, o in g:
...     print(type(s))
...
<class 'rdflib.term.BNode'>
<class 'rdflib.term.BNode'>
<class 'rdflib.term.BNode'>
<class 'rdflib.term.BNode'>
```

```
>>> for s, p, o in g.triples((None, RDF.object, None)):
...     print(o)
...
Conjunctive Graph
>>> g.remove((statementId, RDF.type, RDF.Statement))
>>> print(len(g))
3
```

`None` terms in calls to *`triples()`* can be thought of as "open variables".

Graph support set-theoretic operators, you can add/subtract graphs, as well as intersection (with multiplication operator g1*g2) and xor (g1 ^ g2).

Note that BNode IDs are kept when doing set-theoretic operations, this may or may not be what you want. Two named graphs within the same application probably want share BNode IDs, two graphs with data from different sources probably not. If your BNode IDs are all generated by RDFLib they are UUIDs and unique.

```
>>> g1 = Graph()
>>> g2 = Graph()
>>> u = URIRef("http://example.com/foo")
>>> g1.add([u, RDFS.label, Literal("foo")])
>>> g1.add([u, RDFS.label, Literal("bar")])
>>> g2.add([u, RDFS.label, Literal("foo")])
>>> g2.add([u, RDFS.label, Literal("bing")])
>>> len(g1 + g2)  # adds bing as label
3
>>> len(g1 - g2)  # removes foo
1
>>> len(g1 * g2)  # only foo
1
>>> g1 += g2  # now g1 contains everything
```

Graph Aggregation - ConjunctiveGraphs and ReadOnlyGraphAggregate within the same store:

```
>>> store = plugin.get("IOMemory", Store)()
>>> g1 = Graph(store)
>>> g2 = Graph(store)
>>> g3 = Graph(store)
>>> stmt1 = BNode()
>>> stmt2 = BNode()
>>> stmt3 = BNode()
>>> g1.add((stmt1, RDF.type, RDF.Statement))
>>> g1.add((stmt1, RDF.subject,
```

(continues on next page)

(continued from previous page)

```
...     URIRef('http://rdflib.net/store/ConjunctiveGraph')))
>>> g1.add((stmt1, RDF.predicate, RDFS.label))
>>> g1.add((stmt1, RDF.object, Literal('Conjunctive Graph')))
>>> g2.add((stmt2, RDF.type, RDF.Statement))
>>> g2.add((stmt2, RDF.subject,
...     URIRef('http://rdflib.net/store/ConjunctiveGraph')))
>>> g2.add((stmt2, RDF.predicate, RDF.type))
>>> g2.add((stmt2, RDF.object, RDFS.Class))
>>> g3.add((stmt3, RDF.type, RDF.Statement))
>>> g3.add((stmt3, RDF.subject,
...     URIRef('http://rdflib.net/store/ConjunctiveGraph')))
>>> g3.add((stmt3, RDF.predicate, RDFS.comment))
>>> g3.add((stmt3, RDF.object, Literal(
...     'The top-level aggregate graph - The sum ' +
...     'of all named graphs within a Store')))
>>> len(list(ConjunctiveGraph(store).subjects(RDF.type, RDF.Statement)))
3
>>> len(list(ReadOnlyGraphAggregate([g1,g2]).subjects(
...     RDF.type, RDF.Statement)))
2
```

ConjunctiveGraphs have a *quads()* method which returns quads instead of triples, where the fourth item is the Graph (or subclass thereof) instance in which the triple was asserted:

```
>>> uniqueGraphNames = set(
...     [graph.identifier for s, p, o, graph in ConjunctiveGraph(store
...     ).quads((None, RDF.predicate, None))])
>>> len(uniqueGraphNames)
3
>>> unionGraph = ReadOnlyGraphAggregate([g1, g2])
>>> uniqueGraphNames = set(
...     [graph.identifier for s, p, o, graph in unionGraph.quads(
...     (None, RDF.predicate, None))])
>>> len(uniqueGraphNames)
2
```

Parsing N3 from a string

```
>>> g2 = Graph()
>>> src = '''
... @prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
... @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
... [ a rdf:Statement ;
...   rdf:subject <http://rdflib.net/store#ConjunctiveGraph>;
...   rdf:predicate rdfs:label;
...   rdf:object "Conjunctive Graph" ] .
... '''
>>> g2 = g2.parse(data=src, format="n3")
>>> print(len(g2))
4
```

Using Namespace class:

```
>>> RDFLib = Namespace("http://rdflib.net/")
>>> RDFLib.ConjunctiveGraph
rdflib.term.URIRef('http://rdflib.net/ConjunctiveGraph')
```

(continues on next page)

(continued from previous page)

```
>>> RDFLib["Graph"]
rdflib.term.URIRef('http://rdflib.net/Graph')
```

class rdflib.graph.**Graph**(*store='default'*, *identifier=None*, *namespace_manager=None*, *base=None*)

Bases: *rdflib.term.Node*

An RDF Graph

The constructor accepts one argument, the "store" that will be used to store the graph data (see the "store" package for stores currently shipped with rdflib).

Stores can be context-aware or unaware. Unaware stores take up (some) less space but cannot support features that require context, such as true merging/demerging of sub-graphs and provenance.

The Graph constructor can take an identifier which identifies the Graph by name. If none is given, the graph is assigned a BNode for its identifier.

For more on named graphs, see: http://www.w3.org/2004/03/trix/

__add__(*other*)
Set-theoretic union BNode IDs are not changed.

__and__(*other*)
Set-theoretic intersection. BNode IDs are not changed.

__cmp__(*other*)

__contains__(*triple*)
Support for 'triple in graph' syntax

__dict__ = mappingproxy({'__module__': 'rdflib.graph', '__doc__': 'An RDF Graph\n\n '

__eq__(*other*)
Return self==value.

__ge__(*other*)
Return self>=value.

__getitem__(*item*)
A graph can be "sliced" as a shortcut for the triples method The python slice syntax is (ab)used for specifying triples. A generator over matches is returned, the returned tuples include only the parts not given

```
>>> import rdflib
>>> g = rdflib.Graph()
>>> g.add((rdflib.URIRef("urn:bob"), rdflib.RDFS.label, rdflib.Literal("Bob
↪")))
```

```
>>> list(g[rdflib.URIRef("urn:bob")]) # all triples about bob
[(rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#label'), rdflib.
↪term.Literal('Bob'))]
```

```
>>> list(g[:rdflib.RDFS.label]) # all label triples
[(rdflib.term.URIRef('urn:bob'), rdflib.term.Literal('Bob'))]
```

```
>>> list(g[::rdflib.Literal("Bob")]) # all triples with bob as object
[(rdflib.term.URIRef('urn:bob'), rdflib.term.URIRef('http://www.w3.org/2000/
↪01/rdf-schema#label'))]
```

Combined with SPARQL paths, more complex queries can be written concisely:

Name of all Bobs friends:

g[bob : FOAF.knows/FOAF.name]

Some label for Bob:

g[bob : DC.title|FOAF.name|RDFS.label]

All friends and friends of friends of Bob

g[bob : FOAF.knows * “+”]

etc.

New in version 4.0.

__gt__(*other*)
Return self>value.

__hash__()
Return hash(self).

__iadd__(*other*)
Add all triples in Graph other to Graph. BNode IDs are not changed.

__init__(*store='default'*, *identifier=None*, *namespace_manager=None*, *base=None*)
Initialize self. See help(type(self)) for accurate signature.

__isub__(*other*)
Subtract all triples in Graph other from Graph. BNode IDs are not changed.

__iter__()
Iterates over all triples in the store

__le__(*other*)
Return self<=value.

__len__()
Returns the number of triples in the graph

If context is specified then the number of triples in the context is returned instead.

__lt__(*other*)
Return self<value.

__module__ = 'rdflib.graph'

__mul__(*other*)
Set-theoretic intersection. BNode IDs are not changed.

__or__(*other*)
Set-theoretic union BNode IDs are not changed.

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__str__()
Return str(self).

__sub__(*other*)
Set-theoretic difference. BNode IDs are not changed.

__weakref__
list of weak references to the object (if defined)

__xor__(*other*)
Set-theoretic XOR. BNode IDs are not changed.

absolutize(*uri*, *defrag=1*)
Turn uri into an absolute URI if it's not one already

add(*triple*)
Add a triple with self as context

addN(*quads*)
Add a sequence of triple with context

all_nodes()

bind(*prefix*, *namespace*, *override=True*, *replace=False*)
Bind prefix to namespace

If override is True will bind namespace to given prefix even if namespace was already bound to a different prefix.

if replace, replace any existing prefix with the new namespace

for example: graph.bind("foaf", "http://xmlns.com/foaf/0.1/")

close(*commit_pending_transaction=False*)
Close the graph store

Might be necessary for stores that require closing a connection to a database or releasing some resource.

collection(*identifier*)
Create a new `Collection` instance.

Parameters:

- `identifier`: a URIRef or BNode instance.

Example:

```
>>> graph = Graph()
>>> uri = URIRef("http://example.org/resource")
>>> collection = graph.collection(uri)
>>> assert isinstance(collection, Collection)
>>> assert collection.uri is uri
>>> assert collection.graph is graph
>>> collection += [ Literal(1), Literal(2) ]
```

comment(*subject*, *default=''*)
Query for the RDFS.comment of the subject

Return default if no comment exists

commit()
Commits active transactions

compute_qname(*uri*, *generate=True*)

connected()
Check if the Graph is connected

The Graph is considered undirectional.

Performs a search on the Graph, starting from a random node. Then iteratively goes depth-first through the triplets where the node is subject and object. Return True if all nodes have been visited and False if it cannot continue and there are still unvisited nodes left.

de_skolemize(*new_graph=None*, *uriref=None*)

destroy(*configuration*)
Destroy the store identified by *configuration* if supported

property identifier

isomorphic(*other*)
does a very basic check if these graphs are the same If no BNodes are involved, this is accurate.

See rdflib.compare for a correct implementation of isomorphism checks

items(*list*)
Generator over all items in the resource specified by list

list is an RDF collection.

label(*subject*, *default=''*)
Query for the RDFS.label of the subject

Return default if no label exists or any label if multiple exist.

load(*source*, *publicID=None*, *format='xml'*)

n3()
return an n3 identifier for the Graph

property namespace_manager
this graph's namespace-manager

namespaces()
Generator over all the prefix, namespace tuples

objects(*subject=None*, *predicate=None*)
A generator of objects with the given subject and predicate

open(*configuration*, *create=False*)
Open the graph store

Might be necessary for stores that require opening a connection to a database or acquiring some resource.

parse(*source=None*, *publicID=None*, *format=None*, *location=None*, *file=None*, *data=None*, ***args*)
Parse source adding the resulting triples to the Graph.

The source is specified using one of source, location, file or data.

Parameters

- *source*: An InputSource, file-like object, or string. In the case of a string the string is the location of the source.
- *location*: A string indicating the relative or absolute URL of the source. Graph's absolutize method is used if a relative location is specified.
- *file*: A file-like object.
- *data*: A string containing the data to be parsed.
- *format*: Used if format can not be determined from source. Defaults to rdf/xml. Format support can be extended with plugins, but "xml", "n3", "nt" & "trix" are built in.

- *publicID*: the logical URI to use as the document base. If None specified the document location is used (at least in the case where there is a document location).

Returns

- self, the graph instance.

Examples:

```
>>> my_data = '''
... <rdf:RDF
...   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
...   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
... >
...   <rdf:Description>
...     <rdfs:label>Example</rdfs:label>
...     <rdfs:comment>This is really just an example.</rdfs:comment>
...   </rdf:Description>
... </rdf:RDF>
... '''
>>> import tempfile
>>> fd, file_name = tempfile.mkstemp()
>>> f = os.fdopen(fd, "w")
>>> dummy = f.write(my_data)  # Returns num bytes written
>>> f.close()
```

```
>>> g = Graph()
>>> result = g.parse(data=my_data, format="application/rdf+xml")
>>> len(g)
2
```

```
>>> g = Graph()
>>> result = g.parse(location=file_name, format="application/rdf+xml")
>>> len(g)
2
```

```
>>> g = Graph()
>>> with open(file_name, "r") as f:
...     result = g.parse(f, format="application/rdf+xml")
>>> len(g)
2
```

```
>>> os.remove(file_name)
```

predicate_objects(*subject=None*)

A generator of (predicate, object) tuples for the given subject

predicates(*subject=None*, *object=None*)

A generator of predicates with the given subject and object

preferredLabel(*subject*, *lang=None*, *default=None*, *labelProperties=rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'), rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#label')*)

Find the preferred label for subject.

By default prefers skos:prefLabels over rdfs:labels. In case at least one prefLabel is found returns those, else returns labels. In case a language string (e.g., “en”, “de” or even “” for no lang-tagged literals) is given, only such labels will be considered.

Return a list of (labelProp, label) pairs, where labelProp is either skos:prefLabel or rdfs:label.

```
>>> from rdflib import ConjunctiveGraph, URIRef, RDFS, Literal
>>> from rdflib.namespace import SKOS
>>> from pprint import pprint
>>> g = ConjunctiveGraph()
>>> u = URIRef("http://example.com/foo")
>>> g.add([u, RDFS.label, Literal("foo")])
>>> g.add([u, RDFS.label, Literal("bar")])
>>> pprint(sorted(g.preferredLabel(u)))
[(rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#label'),
  rdflib.term.Literal('bar')),
 (rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#label'),
  rdflib.term.Literal('foo'))]
>>> g.add([u, SKOS.prefLabel, Literal("bla")])
>>> pprint(g.preferredLabel(u))
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('bla'))]
>>> g.add([u, SKOS.prefLabel, Literal("blubb", lang="en")])
>>> sorted(g.preferredLabel(u))
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('bla')),
  (rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('blubb', lang='en'))]
>>> g.preferredLabel(u, lang="")
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('bla'))]
>>> pprint(g.preferredLabel(u, lang="en"))
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('blubb', lang='en'))]
```

qname(*uri*)

query(*query_object*, *processor='sparql'*, *result='sparql'*, *initNs=None*, *initBindings=None*, *use_store_provided=True*, ***kwargs*)

Query this graph.

A type of 'prepared queries' can be realised by providing initial variable bindings with initBindings

Initial namespaces are used to resolve prefixes used in the query, if none are given, the namespaces from the graph's namespace manager are used.

Returntype rdflib.query.QueryResult

remove(*triple*)

Remove a triple from the graph

If the triple does not provide a context attribute, removes the triple from all contexts.

resource(*identifier*)

Create a new `Resource` instance.

Parameters:

- `identifier`: a URIRef or BNode instance.

Example:

```
>>> graph = Graph()
>>> uri = URIRef("http://example.org/resource")
>>> resource = graph.resource(uri)
```

(continues on next page)

(continued from previous page)

```
>>> assert isinstance(resource, Resource)
>>> assert resource.identifier is uri
>>> assert resource.graph is graph
```

rollback()
Rollback active transactions

seq(*subject*)
Check if subject is an rdf:Seq

If yes, it returns a Seq class instance, None otherwise.

serialize(*destination=None*, *format='xml'*, *base=None*, *encoding=None*, ***args*)
Serialize the Graph to destination

If destination is None serialize method returns the serialization as a string. Format defaults to xml (AKA rdf/xml).

Format support can be extended with plugins, but "xml", "n3", "turtle", "nt", "pretty-xml", "trix", "trig" and "nquads" are built in.

set(*triple*)
Convenience method to update the value of object

Remove any existing triples for subject and predicate before adding (subject, predicate, object).

skolemize(*new_graph=None*, *bnode=None*, *authority=None*, *basepath=None*)

property store

subject_objects(*predicate=None*)
A generator of (subject, object) tuples for the given predicate

subject_predicates(*object=None*)
A generator of (subject, predicate) tuples for the given object

subjects(*predicate=None*, *object=None*)
A generator of subjects with the given predicate and object

toPython()

transitiveClosure(*func*, *arg*, *seen=None*)
Generates transitive closure of a user-defined function against the graph

```
>>> from rdflib.collection import Collection
>>> g=Graph()
>>> a=BNode("foo")
>>> b=BNode("bar")
>>> c=BNode("baz")
>>> g.add((a,RDF.first,RDF.type))
>>> g.add((a,RDF.rest,b))
>>> g.add((b,RDF.first,RDFS.label))
>>> g.add((b,RDF.rest,c))
>>> g.add((c,RDF.first,RDFS.comment))
>>> g.add((c,RDF.rest,RDF.nil))
>>> def topList(node,g):
...    for s in g.subjects(RDF.rest, node):
...       yield s
>>> def reverseList(node,g):
...    for f in g.objects(node, RDF.first):
...       print(f)
```

(continues on next page)

(continued from previous page)

```
...     for s in g.subjects(RDF.rest, node):
...         yield s
```

```
>>> [rt for rt in g.transitiveClosure(
...     topList,RDF.nil)]
[rdflib.term.BNode('baz'),
 rdflib.term.BNode('bar'),
 rdflib.term.BNode('foo')]
```

```
>>> [rt for rt in g.transitiveClosure(
...     reverseList,RDF.nil)]
http://www.w3.org/2000/01/rdf-schema#comment
http://www.w3.org/2000/01/rdf-schema#label
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
[rdflib.term.BNode('baz'),
 rdflib.term.BNode('bar'),
 rdflib.term.BNode('foo')]
```

transitive_objects(*subject*, *property*, *remember=None*)
Transitively generate objects for the `property` relationship

Generated objects belong to the depth first transitive closure of the `property` relationship starting at `subject`.

transitive_subjects(*predicate*, *object*, *remember=None*)
Transitively generate objects for the `property` relationship

Generated objects belong to the depth first transitive closure of the `property` relationship starting at `subject`.

triples(*triple*)
Generator over the triple store

Returns triples that match the given triple pattern. If triple pattern does not provide a context, all contexts will be searched.

triples_choices(*triple*, *context=None*)

update(*update_object*, *processor='sparql'*, *initNs=None*, *initBindings=None*, *use_store_provided=True*, ***kwargs*)
Update this graph with the given update query.

value(*subject=None*, *predicate=rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#value')*, *object=None*, *default=None*, *any=True*)
Get a value for a pair of two criteria

Exactly one of subject, predicate, object must be None. Useful if one knows that there may only be one value.

It is one of those situations that occur a lot, hence this 'macro' like utility

Parameters: subject, predicate, object – exactly one must be None default – value to be returned if no values found any – if True, return any value in the case there is more than one, else, raise UniquenessError

class rdflib.graph.**ConjunctiveGraph**(*store='default'*, *identifier=None*, *default_graph_base=None*)
Bases: *rdflib.graph.Graph*

A ConjunctiveGraph is an (unnamed) aggregation of all the named graphs in a store.

It has a `default` graph, whose name is associated with the graph throughout its life. *__init__()* can take an identifier to use as the name of this default graph or it will assign a BNode.

All methods that add triples work against this default graph.

All queries are carried out against the union of all graphs.

__contains__(*triple_or_quad*)
Support for 'triple/quad in graph' syntax

__init__(*store='default'*, *identifier=None*, *default_graph_base=None*)
Initialize self. See help(type(self)) for accurate signature.

__len__()
Number of triples in the entire conjunctive graph

__module__ = 'rdflib.graph'

__reduce__()
Helper for pickle.

__str__()
Return str(self).

add(*triple_or_quad*)
Add a triple or quad to the store.

if a triple is given it is added to the default context

addN(*quads*)
Add a sequence of triples with context

context_id(*uri*, *context_id=None*)
URI#context

contexts(*triple=None*)
Iterate over all contexts in the graph

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

get_context(*identifier*, *quoted=False*, *base=None*)
Return a context graph for the given identifier

identifier must be a URIRef or BNode.

parse(*source=None*, *publicID=None*, *format='xml'*, *location=None*, *file=None*, *data=None*, ***args*)
Parse source adding the resulting triples to its own context (sub graph of this graph).

See *rdflib.graph.Graph.parse()* for documentation on arguments.

Returns

The graph into which the source was parsed. In the case of n3 it returns the root context.

quads(*triple_or_quad=None*)
Iterate over all the quads in the entire conjunctive graph

remove(*triple_or_quad*)
Removes a triple or quads

if a triple is given it is removed from all contexts

a quad is removed from the given context only

remove_context(*context*)
Removes the given context from the graph

triples(*triple_or_quad*, *context=None*)
Iterate over all the triples in the entire conjunctive graph

For legacy reasons, this can take the context to query either as a fourth element of the quad, or as the explicit context keyword parameter. The kw param takes precedence.

triples_choices(*triple*, *context=None*)
Iterate over all the triples in the entire conjunctive graph

class rdflib.graph.**QuotedGraph**(*store*, *identifier*)
Bases: *rdflib.graph.Graph*

Quoted Graphs are intended to implement Notation 3 formulae. They are associated with a required identifier that the N3 parser *must* provide in order to maintain consistent formulae identification for scenarios such as implication and other such processing.

__init__(*store*, *identifier*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.graph'

__reduce__()
Helper for pickle.

__str__()
Return str(self).

add(*triple*)
Add a triple with self as context

addN(*quads*)
Add a sequence of triple with context

n3()
Return an n3 identifier for the Graph

class rdflib.graph.**Seq**(*graph*, *subject*)
Bases: object

Wrapper around an RDF Seq resource

It implements a container type in Python with the order of the items returned corresponding to the Seq content. It is based on the natural ordering of the predicate names _1, _2, _3, etc, which is the 'implementation' of a sequence in RDF terms.

__dict__ = mappingproxy({'__module__': 'rdflib.graph', '__doc__': "Wrapper around an

__getitem__(*index*)
Item given by index from the Seq

__init__(*graph*, *subject*)
Parameters:

- **graph:** the graph containing the Seq
- **subject:** the subject of a Seq. Note that the init does not check whether this is a Seq, this is done in whoever creates this instance!

__iter__()
Generator over the items in the Seq

__len__()
Length of the Seq

__module__ = 'rdflib.graph'

__weakref__
list of weak references to the object (if defined)

toPython()

exception rdflib.graph.**ModificationException**
Bases: Exception

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.graph'

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

class rdflib.graph.**Dataset**(*store='default'*, *default_union=False*, *default_graph_base=None*)
Bases: *rdflib.graph.ConjunctiveGraph*

RDF 1.1 Dataset. Small extension to the Conjunctive Graph: - the primary term is graphs in the datasets and not contexts with quads, so there is a separate method to set/retrieve a graph in a dataset and operate with graphs - graphs cannot be identified with blank nodes - added a method to directly add a single quad

Examples of usage:

```
>>> # Create a new Dataset
>>> ds = Dataset()
>>> # simple triples goes to default graph
>>> ds.add((URIRef("http://example.org/a"),
...    URIRef("http://www.example.org/b"),
...    Literal("foo")))
>>>
>>> # Create a graph in the dataset, if the graph name has already been
>>> # used, the corresponding graph will be returned
>>> # (ie, the Dataset keeps track of the constituent graphs)
>>> g = ds.graph(URIRef("http://www.example.com/gr"))
>>>
>>> # add triples to the new graph as usual
>>> g.add(
...     (URIRef("http://example.org/x"),
...     URIRef("http://example.org/y"),
...     Literal("bar")) )
>>> # alternatively: add a quad to the dataset -> goes to the graph
>>> ds.add(
...     (URIRef("http://example.org/x"),
...     URIRef("http://example.org/z"),
...     Literal("foo-bar"),g) )
>>>
>>> # querying triples return them all regardless of the graph
>>> for t in ds.triples((None,None,None)):
...     print(t)
(rdflib.term.URIRef("http://example.org/a"),
 rdflib.term.URIRef("http://www.example.org/b"),
 rdflib.term.Literal("foo"))
(rdflib.term.URIRef("http://example.org/x"),
 rdflib.term.URIRef("http://example.org/z"),
 rdflib.term.Literal("foo-bar"))
(rdflib.term.URIRef("http://example.org/x"),
```

(continues on next page)

(continued from previous page)

```
 rdflib.term.URIRef("http://example.org/y"),
 rdflib.term.Literal("bar"))
>>>
>>> # querying quads return quads; the fourth argument can be unrestricted
>>> # or restricted to a graph
>>> for q in ds.quads((None, None, None, None)):
...     print(q)
(rdflib.term.URIRef("http://example.org/a"),
 rdflib.term.URIRef("http://www.example.org/b"),
 rdflib.term.Literal("foo"),
 None)
(rdflib.term.URIRef("http://example.org/x"),
 rdflib.term.URIRef("http://example.org/y"),
 rdflib.term.Literal("bar"),
 rdflib.term.URIRef("http://www.example.com/gr"))
(rdflib.term.URIRef("http://example.org/x"),
 rdflib.term.URIRef("http://example.org/z"),
 rdflib.term.Literal("foo-bar"),
 rdflib.term.URIRef("http://www.example.com/gr"))
>>>
>>> for q in ds.quads((None,None,None,g)):
...     print(q)
(rdflib.term.URIRef("http://example.org/x"),
 rdflib.term.URIRef("http://example.org/y"),
 rdflib.term.Literal("bar"),
 rdflib.term.URIRef("http://www.example.com/gr"))
(rdflib.term.URIRef("http://example.org/x"),
 rdflib.term.URIRef("http://example.org/z"),
 rdflib.term.Literal("foo-bar"),
 rdflib.term.URIRef("http://www.example.com/gr"))
>>> # Note that in the call above -
>>> # ds.quads((None,None,None,"http://www.example.com/gr"))
>>> # would have been accepted, too
>>>
>>> # graph names in the dataset can be queried:
>>> for c in ds.graphs():
...     print(c)  # doctest:
DEFAULT
http://www.example.com/gr
>>> # A graph can be created without specifying a name; a skolemized genid
>>> # is created on the fly
>>> h = ds.graph()
>>> for c in ds.graphs():
...     print(c)
DEFAULT
http://rdlib.net/.well-known/genid/rdflib/N...
http://www.example.com/gr
>>> # Note that the Dataset.graphs() call returns names of empty graphs,
>>> # too. This can be restricted:
>>> for c in ds.graphs(empty=False):
...     print(c)
DEFAULT
http://www.example.com/gr
>>>
>>> # a graph can also be removed from a dataset via ds.remove_graph(g)
```

New in version 4.0.

__init__(*store='default'*, *default_union=False*, *default_graph_base=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.graph'

__str__()
Return str(self).

add_graph(*g*)
alias of graph for consistency

contexts(*triple=None*)
Iterate over all contexts in the graph

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

graph(*identifier=None*, *base=None*)

graphs(*triple=None*)
Iterate over all contexts in the graph

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

parse(*source=None*, *publicID=None*, *format='xml'*, *location=None*, *file=None*, *data=None*, ***args*)
Parse source adding the resulting triples to its own context (sub graph of this graph).

See *rdflib.graph.Graph.parse()* for documentation on arguments.

Returns

The graph into which the source was parsed. In the case of n3 it returns the root context.

quads(*quad*)
Iterate over all the quads in the entire conjunctive graph

remove_graph(*g*)

exception rdflib.graph.**UnSupportedAggregateOperation**
Bases: Exception

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.graph'

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

class rdflib.graph.**ReadOnlyGraphAggregate**(*graphs*, *store='default'*)
Bases: *rdflib.graph.ConjunctiveGraph*

Utility class for treating a set of graphs as a single graph

Only read operations are supported (hence the name). Essentially a ConjunctiveGraph over an explicit subset of the entire store.

__cmp__(*other*)

__contains__(*triple_or_quad*)
Support for 'triple/quad in graph' syntax

__hash__()
Return hash(self).

__iadd__(*other*)
Add all triples in Graph other to Graph. BNode IDs are not changed.

__init__(*graphs*, *store='default'*)
Initialize self. See help(type(self)) for accurate signature.

__isub__(*other*)
Subtract all triples in Graph other from Graph. BNode IDs are not changed.

__len__()
Number of triples in the entire conjunctive graph

__module__ = 'rdflib.graph'

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

absolutize(*uri*, *defrag=1*)
Turn uri into an absolute URI if it's not one already

add(*triple*)
Add a triple or quad to the store.

if a triple is given it is added to the default context

addN(*quads*)
Add a sequence of triples with context

bind(*prefix*, *namespace*, *override=True*)
Bind prefix to namespace

If override is True will bind namespace to given prefix even if namespace was already bound to a different prefix.

if replace, replace any existing prefix with the new namespace

for example: graph.bind("foaf", "http://xmlns.com/foaf/0.1/")

close()
Close the graph store

Might be necessary for stores that require closing a connection to a database or releasing some resource.

commit()
Commits active transactions

compute_qname(*uri*, *generate=True*)

destroy(*configuration*)
Destroy the store identified by *configuration* if supported

n3()
return an n3 identifier for the Graph

namespaces()
Generator over all the prefix, namespace tuples

open(*configuration*, *create=False*)
Open the graph store

Might be necessary for stores that require opening a connection to a database or acquiring some resource.

parse(*source*, *publicID=None*, *format='xml'*, ***args*)
Parse source adding the resulting triples to its own context (sub graph of this graph).

See *rdflib.graph.Graph.parse()* for documentation on arguments.

Returns

The graph into which the source was parsed. In the case of n3 it returns the root context.

qname(*uri*)

quads(*triple*)
Iterate over all the quads in the entire aggregate graph

remove(*triple*)
Removes a triple or quads

if a triple is given it is removed from all contexts

a quad is removed from the given context only

rollback()
Rollback active transactions

triples(*triple*)
Iterate over all the triples in the entire conjunctive graph

For legacy reasons, this can take the context to query either as a fourth element of the quad, or as the explicit context keyword parameter. The kw param takes precedence.

triples_choices(*triple*, *context=None*)
Iterate over all the triples in the entire conjunctive graph

class rdflib.graph.**BatchAddGraph**(*graph*, *batch_size=1000*, *batch_addn=False*)
Bases: object

Wrapper around graph that turns calls to *add()* (and optionally, *addN()*) into calls to *addN()*.

Parameters

- *graph*: The graph to wrap
- *batch_size*: The maximum number of triples to buffer before passing to *graph*'s *addN*
- *batch_addn*: If True, then even calls to *addN* will be batched according to *batch_size*

Variables

- ***graph*** – The wrapped graph
- **count** – The number of triples buffered since initaialization or the last call to *reset()*
- **batch** – The current buffer of triples

__dict__ = mappingproxy({'__module__': 'rdflib.graph', '__doc__': "\n Wrapper around

__enter__()

__exit__(**exc*)

__init__(*graph*, *batch_size=1000*, *batch_addn=False*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.graph'

__weakref__
list of weak references to the object (if defined)

add(*triple_or_quad*)
Add a triple to the buffer

Parameters **triple** – The triple to add

addN(*quads*)

reset()
Manually clear the buffered triples and reset the count to zero

rdflib.namespace module

Namespace Utilities

RDFLib provides mechanisms for managing Namespaces.

In particular, there is a *Namespace* class that takes as its argument the base URI of the namespace.

```
>>> from rdflib.namespace import Namespace
>>> owl = Namespace('http://www.w3.org/2002/07/owl#')
```

Fully qualified URIs in the namespace can be constructed either by attribute or by dictionary access on Namespace instances:

```
>>> owl.seeAlso
rdflib.term.URIRef(u'http://www.w3.org/2002/07/owl#seeAlso')
>>> owl['seeAlso']
rdflib.term.URIRef(u'http://www.w3.org/2002/07/owl#seeAlso')
```

Automatic handling of unknown predicates

As a programming convenience, a namespace binding is automatically created when *rdflib.term.URIRef* predicates are added to the graph.

Importable namespaces

The following namespaces are available by directly importing from rdflib:

- RDF
- RDFS
- OWL
- XSD
- FOAF
- SKOS
- DOAP
- DC
- DCTERMS
- VOID

```
>>> from rdflib import OWL
>>> OWL.seeAlso
rdflib.term.URIRef(u'http://www.w3.org/2002/07/owl#seeAlso')
```

rdflib.namespace.**is_ncname**(*name*)

rdflib.namespace.**split_uri**(*uri*, *split_start=['Ll', 'Lu', 'Lo', 'Lt', 'Nl', 'Nd']*)

class rdflib.namespace.**Namespace**
Bases: str

Utility class for quickly generating URIRefs with a common prefix

```
>>> from rdflib import Namespace
>>> n = Namespace("http://example.org/")
>>> n.Person # as attribute
rdflib.term.URIRef(u'http://example.org/Person')
>>> n['first-name'] # as item - for things that are not valid python identifiers
rdflib.term.URIRef(u'http://example.org/first-name')
```

__dict__ = mappingproxy({'__module__': 'rdflib.namespace', '__doc__': '\n Utility cl

__getattr__(*name*)

__getitem__(*key*, *default=None*)
Return self[key].

__module__ = 'rdflib.namespace'

static __new__(*cls*, *value*)
Create and return a new object. See help(type) for accurate signature.

__repr__()
Return repr(self).

__weakref__
list of weak references to the object (if defined)

term(*name*)

property title
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

class rdflib.namespace.**ClosedNamespace**(*uri*, *terms*)
Bases: object

A namespace with a closed list of members

Trying to create terms not listen is an error

__dict__ = mappingproxy({'__module__': 'rdflib.namespace', '__doc__': '\n A namespac

__getattr__(*name*)

__getitem__(*key*, *default=None*)

__init__(*uri*, *terms*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.namespace'

__repr__()
Return repr(self).

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

term(*name*)

class rdflib.namespace.**NamespaceManager**(*graph*)
Bases: object

Class for managing prefix => namespace mappings

Sample usage from FuXi ...

```
ruleStore = N3RuleStore(additionalBuiltins=additionalBuiltins)
nsMgr = NamespaceManager(Graph(ruleStore))
ruleGraph = Graph(ruleStore,namespace_manager=nsMgr)
```

and ...

```
>>> import rdflib
>>> from rdflib import Graph
>>> from rdflib.namespace import Namespace, NamespaceManager
>>> exNs = Namespace('http://example.com/')
>>> namespace_manager = NamespaceManager(Graph())
>>> namespace_manager.bind('ex', exNs, override=False)
>>> g = Graph()
>>> g.namespace_manager = namespace_manager
>>> all_ns = [n for n in g.namespace_manager.namespaces()]
>>> assert ('ex', rdflib.term.URIRef('http://example.com/')) in all_ns
>>>
```

__dict__ = mappingproxy({'__module__': 'rdflib.namespace', '__doc__': "\n\n Class fo

__init__(*graph*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.namespace'

__weakref__
list of weak references to the object (if defined)

absolutize(*uri*, *defrag=1*)

bind(*prefix*, *namespace*, *override=True*, *replace=False*)
bind a given namespace to the prefix

if override, rebind, even if the given namespace is already bound to another prefix.

if replace, replace any existing prefix with the new namespace

compute_qname(*uri*, *generate=True*)

compute_qname_strict(*uri*, *generate=True*)

namespaces()

normalizeUri(*rdfTerm*)
Takes an RDF Term and 'normalizes' it into a QName (using the registered prefix) or (unlike compute_qname) the Notation 3 form for URIs: <...URI...>

qname(*uri*)

qname_strict(*uri*)

reset()

property store

rdflib.parser module

Parser plugin interface.

This module defines the parser plugin interface and contains other related parser support code.

The module is mainly useful for those wanting to write a parser that can plugin to rdflib. If you are wanting to invoke a parser you likely want to do so through the Graph class parse method.

class rdflib.parser.**Parser**
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.parser', '__init__': <function Parser

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.parser'

__weakref__
list of weak references to the object (if defined)

parse(*source*, *sink*)

class rdflib.parser.**InputSource**(*system_id=None*)
Bases: xml.sax.xmlreader.InputSource, object

TODO:

__init__(*system_id=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.parser'

close()

class rdflib.parser.**StringInputSource**(*value*, *system_id=None*)
Bases: *rdflib.parser.InputSource*

TODO:

__init__(*value*, *system_id=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.parser'

class rdflib.parser.**URLInputSource**(*system_id=None*, *format=None*)
Bases: *rdflib.parser.InputSource*

TODO:

__init__(*system_id=None*, *format=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.parser'

__repr__()
Return repr(self).

class rdflib.parser.**FileInputSource**(*file*)
Bases: *rdflib.parser.InputSource*

__init__(*file*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.parser'

__repr__()
Return repr(self).

rdflib.paths module

This module implements the SPARQL 1.1 Property path operators, as defined in:

http://www.w3.org/TR/sparql11-query/#propertypaths

In SPARQL the syntax is as follows:

Syntax	Matches
iri	An IRI. A path of length one.
^elt	Inverse path (object to subject).
elt1 / elt2	A sequence path of elt1 followed by elt2.
elt1 | elt2	A alternative path of elt1 or elt2 (all possibilities are tried).
elt*	A path that connects the subject and object of the path by zero or more matches of elt.
elt+	A path that connects the subject and object of the path by one or more matches of elt.
elt?	A path that connects the subject and object of the path by zero or one matches of elt.
!iri or !(iri_1| ... |iri_n)	Negated property set. An IRI which is not one of $iri_1 \ldots iri_n$. !iri is short for !(iri).
!^iri or !(^iri_1| ...|^iri_n)	Negated property set where the excluded matches are based on reversed path. That is, not one of $iri_1 \ldots iri_n$ as reverse paths. !^iri is short for !(^iri).
!(iri_1| ...|iri_j|^iri_{j+1}|... |^iri_n)|	A combination of forward and reverse properties in a negated property set.
(elt)	A group path elt, brackets control precedence.

This module is used internally be the SPARQL engine, but they property paths can also be used to query RDFLib Graphs directly.

Where possible the SPARQL syntax is mapped to python operators, and property path objects can be constructed from existing URIRefs.

```
>>> from rdflib import Graph, Namespace
```

```
>>> foaf=Namespace('http://xmlns.com/foaf/0.1/')
```

```
>>> ~foaf.knows
Path(~http://xmlns.com/foaf/0.1/knows)
```

```
>>> foaf.knows/foaf.name
Path(http://xmlns.com/foaf/0.1/knows / http://xmlns.com/foaf/0.1/name)
```

```
>>> foaf.name|foaf.givenName
Path(http://xmlns.com/foaf/0.1/name | http://xmlns.com/foaf/0.1/givenName)
```

Modifiers (?, , +) *are done using * (the multiplication operator) and the strings* '', '?', '+', also defined as constants in this file.

```
>>> foaf.knows*OneOrMore
Path(http://xmlns.com/foaf/0.1/knows+)
```

The path objects can also be used with the normal graph methods.

First some example data:

```
>>> g=Graph()
```

```
>>> g=g.parse(data='''
... @prefix : <ex:> .
...
... :a :p1 :c ; :p2 :f .
... :c :p2 :e ; :p3 :g .
... :g :p3 :h ; :p2 :j .
... :h :p3 :a ; :p2 :g .
...
... :q :px :q .
...
... ''', format='n3')
```

```
>>> e=Namespace('ex:')
```

Graph contains: >>> (e.a, e.p1/e.p2, e.e) in g True

Graph generator functions, triples, subjects, objects, etc. :

```
>>> list(g.objects(e.c, (e.p3*OneOrMore)/e.p2))
[rdflib.term.URIRef(u'ex:j'), rdflib.term.URIRef(u'ex:g'),
    rdflib.term.URIRef(u'ex:f')]
```

A more complete set of tests:

```
>>> list(evalPath(g, (None, e.p1/e.p2, None))==[(e.a, e.e)]
True
>>> list(evalPath(g, (e.a, e.p1|e.p2, None))==[(e.a,e.c), (e.a,e.f)]
True
>>> list(evalPath(g, (e.c, ~e.p1, None))) == [ (e.c, e.a) ]
True
>>> list(evalPath(g, (e.a, e.p1*ZeroOrOne, None))) == [(e.a, e.a), (e.a, e.c)]
True
>>> list(evalPath(g, (e.c, e.p3*OneOrMore, None))) == [
...      (e.c, e.g), (e.c, e.h), (e.c, e.a)]
True
>>> list(evalPath(g, (e.c, e.p3*ZeroOrMore, None))) == [(e.c, e.c),
...      (e.c, e.g), (e.c, e.h), (e.c, e.a)]
True
>>> list(evalPath(g, (e.a, -e.p1, None))) == [(e.a, e.f)]
True
>>> list(evalPath(g, (e.a, -(e.p1|e.p2), None))) == []
True
>>> list(evalPath(g, (e.g, -~e.p2, None))) == [(e.g, e.j)]
True
>>> list(evalPath(g, (e.e, ~(e.p1/e.p2), None))) == [(e.e, e.a)]
True
```

(continues on next page)

(continued from previous page)

```
>>> list(evalPath(g, (e.a, e.p1/e.p3/e.p3, None))) == [(e.a, e.h)]
True
```

```
>>> list(evalPath(g, (e.q, e.px*OneOrMore, None)))
[(rdflib.term.URIRef(u'ex:q'), rdflib.term.URIRef(u'ex:q'))]
```

```
>>> list(evalPath(g, (None, e.p1|e.p2, e.c)))
[(rdflib.term.URIRef(u'ex:a'), rdflib.term.URIRef(u'ex:c'))]
```

```
>>> list(evalPath(g, (None, ~e.p1, e.a))) == [ (e.c, e.a) ]
True
>>> list(evalPath(g, (None, e.p1*ZeroOrOne, e.c)))
[(rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:c')),
 (rdflib.term.URIRef(u'ex:a'), rdflib.term.URIRef(u'ex:c'))]
```

```
>>> list(evalPath(g, (None, e.p3*OneOrMore, e.a)))
[(rdflib.term.URIRef(u'ex:h'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:g'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:a'))]
```

```
>>> list(evalPath(g, (None, e.p3*ZeroOrMore, e.a)))
[(rdflib.term.URIRef(u'ex:a'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:h'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:g'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:a'))]
```

```
>>> list(evalPath(g, (None, -e.p1, e.f))) == [(e.a, e.f)]
True
>>> list(evalPath(g, (None, -(e.p1|e.p2), e.c))) == []
True
>>> list(evalPath(g, (None, -~e.p2, e.j))) == [(e.g, e.j)]
True
>>> list(evalPath(g, (None, ~(e.p1/e.p2), e.a))) == [(e.e, e.a)]
True
>>> list(evalPath(g, (None, e.p1/e.p3/e.p3, e.h))) == [(e.a, e.h)]
True
```

```
>>> list(evalPath(g, (e.q, e.px*OneOrMore, None)))
[(rdflib.term.URIRef(u'ex:q'), rdflib.term.URIRef(u'ex:q'))]
```

```
>>> list(evalPath(g, (e.c, (e.p2|e.p3)*ZeroOrMore, e.j)))
[(rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:j'))]
```

No vars specified:

```
>>> sorted(list(evalPath(g, (None, e.p3*OneOrMore, None))))
[(rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:g')),
 (rdflib.term.URIRef(u'ex:c'), rdflib.term.URIRef(u'ex:h')),
 (rdflib.term.URIRef(u'ex:g'), rdflib.term.URIRef(u'ex:a')),
 (rdflib.term.URIRef(u'ex:g'), rdflib.term.URIRef(u'ex:h')),
 (rdflib.term.URIRef(u'ex:h'), rdflib.term.URIRef(u'ex:a'))]
```

New in version 4.0.

class rdflib.paths.**AlternativePath**(**args*)
Bases: *rdflib.paths.Path*

__init__(**args*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.paths'

__repr__()
Return repr(self).

eval(*graph*, *subj=None*, *obj=None*)

n3()

class rdflib.paths.**InvPath**(*arg*)
Bases: *rdflib.paths.Path*

__init__(*arg*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.paths'

__repr__()
Return repr(self).

eval(*graph*, *subj=None*, *obj=None*)

n3()

class rdflib.paths.**MulPath**(*path*, *mod*)
Bases: *rdflib.paths.Path*

__init__(*path*, *mod*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.paths'

__repr__()
Return repr(self).

eval(*graph*, *subj=None*, *obj=None*, *first=True*)

n3()

class rdflib.paths.**NegatedPath**(*arg*)
Bases: *rdflib.paths.Path*

__init__(*arg*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.paths'

__repr__()
Return repr(self).

eval(*graph*, *subj=None*, *obj=None*)

n3()

class rdflib.paths.**Path**
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.paths', 'eval': <function Path.eval>,

__eq__(*other*)
Return self==value.

__ge__(*other*)
Return self>=value.

__gt__(*other*)
Return self>value.

__hash__()
Return hash(self).

__invert__()
inverse path

__le__(*other*)
Return self<=value.

__lt__(*other*)
Return self<value.

__module__ = 'rdflib.paths'

__mul__(*mul*)
cardinality path

__ne__(*other*)
Return self!=value.

__neg__()
negated path

__or__(*other*)
alternative path

__truediv__(*other*)
sequence path

__weakref__
list of weak references to the object (if defined)

eval(*graph*, *subj=None*, *obj=None*)

class rdflib.paths.**PathList**
Bases: list

__dict__ = mappingproxy({'__module__': 'rdflib.paths', '__dict__': <attribute '__dic

__module__ = 'rdflib.paths'

__weakref__
list of weak references to the object (if defined)

class rdflib.paths.**SequencePath**(**args*)
Bases: *rdflib.paths.Path*

__init__(**args*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.paths'

__repr__()
Return repr(self).

eval(*graph*, *subj=None*, *obj=None*)

n3()

rdflib.paths.**evalPath**(*graph*, *t*)

rdflib.paths.**inv_path**(*p*)
inverse path

rdflib.paths.**mul_path**(*p*, *mul*)
cardinality path

rdflib.paths.**neg_path**(*p*)
negated path

rdflib.paths.**path_alternative**(*self*, *other*)
alternative path

rdflib.paths.**path_sequence**(*self*, *other*)
sequence path

rdflib.plugin module

Plugin support for rdf.

There are a number of plugin points for rdf: parser, serializer, store, query processor, and query result. Plugins can be registered either through setuptools entry_points or by calling rdf.plugin.register directly.

If you have a package that uses a setuptools based setup.py you can add the following to your setup:

```
entry_points = {
    'rdf.plugins.parser': [
        'nt =     rdf.plugins.parsers.nt:NTParser',
        ],
    'rdf.plugins.serializer': [
        'nt =     rdf.plugins.serializers.NTSerializer:NTSerializer',
        ],
    }
```

See the setuptools dynamic discovery of services and plugins for more information.

rdflib.plugin.**register**(*name*, *kind*, *module_path*, *class_name*)
Register the plugin for (name, kind). The module_path and class_name should be the path to a plugin class.

rdflib.plugin.**get**(*name*, *kind*)
Return the class for the specified (name, kind). Raises a PluginException if unable to do so.

rdflib.plugin.**plugins**(*name=None*, *kind=None*)
A generator of the plugins.

Pass in name and kind to filter... else leave None to match all.

exception rdflib.plugin.**PluginException**(*msg=None*)
Bases: *rdflib.exceptions.Error*

__module__ = 'rdflib.plugin'

class rdflib.plugin.**Plugin**(*name*, *kind*, *module_path*, *class_name*)
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.plugin', '__init__': <function Plugin

__init__(*name*, *kind*, *module_path*, *class_name*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugin'

__weakref__
list of weak references to the object (if defined)

getClass()

class rdflib.plugin.**PKGPlugin**(*name*, *kind*, *ep*)
Bases: *rdflib.plugin.Plugin*

__init__(*name*, *kind*, *ep*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.plugin'

getClass()

rdflib.query module

class rdflib.query.**Processor**(*graph*)
Bases: object

Query plugin interface.

This module is useful for those wanting to write a query processor that can plugin to rdf. If you are wanting to execute a query you likely want to do so through the Graph class query method.

__dict__ = mappingproxy({'__module__': 'rdflib.query', '__doc__': '\n Query plugin i

__init__(*graph*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.query'

__weakref__
list of weak references to the object (if defined)

query(*strOrQuery*, *initBindings={}*, *initNs={}*, *DEBUG=False*)

class rdflib.query.**Result**(*type_*)
Bases: object

A common class for representing query result.

There is a bit of magic here that makes this appear like different Python objects, depending on the type of result.

If the type is “SELECT”, iterating will yield lists of ResultRow objects

If the type is “ASK”, iterating will yield a single bool (or bool(result) will return the same bool)

If the type is “CONSTRUCT” or “DESCRIBE” iterating will yield the triples.

len(result) also works.

__bool__()

__dict__ = mappingproxy({'__module__': 'rdflib.query', '__doc__': '\n A common class

__eq__(*other*)
Return self==value.

__getattr__(*name*)

__hash__ = None

__init__(*type_*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()

__len__()

__module__ = 'rdflib.query'

__weakref__
list of weak references to the object (if defined)

property bindings
a list of variable bindings as dicts

static parse(*source=None*, *format=None*, *content_type=None*, ***kwargs*)

serialize(*destination=None*, *encoding='utf-8'*, *format='xml'*, ***args*)

class rdflib.query.**ResultParser**
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.query', '__init__': <function ResultP

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.query'

__weakref__
list of weak references to the object (if defined)

parse(*source*, ***kwargs*)
return a Result object

class rdflib.query.**ResultSerializer**(*result*)
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.query', '__init__': <function ResultS

__init__(*result*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.query'

__weakref__
list of weak references to the object (if defined)

serialize(*stream*, *encoding='utf-8'*, ***kwargs*)
return a string properly serialized

exception rdflib.query.**ResultException**
Bases: Exception

__module__ = 'rdflib.query'

__weakref__
list of weak references to the object (if defined)

rdflib.resource module

The *Resource* class wraps a *Graph* and a resource reference (i.e. a *rdflib.term.URIRef* or *rdflib.term.BNode*) to support a resource-oriented way of working with a graph.

It contains methods directly corresponding to those methods of the Graph interface that relate to reading and writing data. The difference is that a Resource also binds a resource identifier, making it possible to work without tracking both the graph and a current subject. This makes for a "resource oriented" style, as compared to the triple orientation of the Graph API.

Resulting generators are also wrapped so that any resource reference values (rdflib.term.URIRef`s and :class:`rdflib.term.BNode`s) are in turn wrapped as Resources. (Note that this

behaviour differs from the corresponding methods in :class:`~rdflib.graph.Graph`, where no such conversion takes place.)

Basic Usage Scenario

Start by importing things we need and define some namespaces:

```
>>> from rdflib import *
>>> FOAF = Namespace("http://xmlns.com/foaf/0.1/")
>>> CV = Namespace("http://purl.org/captsolo/resume-rdf/0.2/cv#")
```

Load some RDF data:

```
>>> graph = Graph().parse(format='n3', data='''
... @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
... @prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
... @prefix foaf: <http://xmlns.com/foaf/0.1/> .
... @prefix cv: <http://purl.org/captsolo/resume-rdf/0.2/cv#> .
...
... @base <http://example.org/> .
...
... </person/some1#self> a foaf:Person;
...     rdfs:comment "Just a Python & RDF hacker."@en;
...     foaf:depiction </images/person/some1.jpg>;
...     foaf:homepage <http://example.net/>;
...     foaf:name "Some Body" .
...
... </images/person/some1.jpg> a foaf:Image;
...     rdfs:label "some 1"@en;
...     rdfs:comment "Just an image"@en;
...     foaf:thumbnail </images/person/some1-thumb.jpg> .
...
... </images/person/some1-thumb.jpg> a foaf:Image .
...
... [] a cv:CV;
...     cv:aboutPerson </person/some1#self>;
...     cv:hasWorkHistory [ cv:employedIn </#company>;
...             cv:startDate "2009-09-04"^^xsd:date ] .
... ''')
```

Create a Resource:

```
>>> person = Resource(
...     graph, URIRef("http://example.org/person/some1#self"))
```

Retrieve some basic facts:

```
>>> person.identifier
rdflib.term.URIRef(u'http://example.org/person/some1#self')

>>> person.value(FOAF.name)
rdflib.term.Literal(u'Some Body')

>>> person.value(RDFS.comment)
rdflib.term.Literal(u'Just a Python & RDF hacker.', lang=u'en')
```

Resources can be sliced (like graphs, but the subject is fixed):

```
>>> for name in person[FOAF.name]:
...     print(name)
Some Body
>>> person[FOAF.name : Literal("Some Body")]
True
```

Resources as unicode are represented by their identifiers as unicode:

```
>>> %(unicode)s(person)
u'Resource(http://example.org/person/some1#self'
```

Resource references are also Resources, so you can easily get e.g. a qname for the type of a resource, like:

```
>>> person.value(RDF.type).qname()
u'foaf:Person'
```

Or for the predicates of a resource:

```
>>> sorted(
...     p.qname() for p in person.predicates()
... )
[u'foaf:depiction', u'foaf:homepage',
 u'foaf:name', u'rdf:type', u'rdfs:comment']
```

Follow relations and get more data from their Resources as well:

```
>>> for pic in person.objects(FOAF.depiction):
...     print(pic.identifier)
...     print(pic.value(RDF.type).qname())
...     print(pic.label())
...     print(pic.comment())
...     print(pic.value(FOAF.thumbnail).identifier)
http://example.org/images/person/some1.jpg
foaf:Image
some 1
Just an image
http://example.org/images/person/some1-thumb.jpg

>>> for cv in person.subjects(CV.aboutPerson):
...     work = list(cv.objects(CV.hasWorkHistory))[0]
...     print(work.value(CV.employedIn).identifier)
...     print(work.value(CV.startDate))
http://example.org/#company
2009-09-04
```

It's just as easy to work with the predicates of a resource:

```
>>> for s, p in person.subject_predicates():
...     print(s.value(RDF.type).qname())
...     print(p.qname())
...     for s, o in p.subject_objects():
...         print(s.value(RDF.type).qname())
...         print(o.value(RDF.type).qname())
cv:CV
cv:aboutPerson
cv:CV
foaf:Person
```

This is useful for e.g. inspection:

```
>>> thumb_ref = URIRef("http://example.org/images/person/some1-thumb.jpg")
>>> thumb = Resource(graph, thumb_ref)
>>> for p, o in thumb.predicate_objects():
...     print(p.qname())
...     print(o.qname())
rdf:type
foaf:Image
```

Similarly, adding, setting and removing data is easy:

```
>>> thumb.add(RDFS.label, Literal("thumb"))
>>> print(thumb.label())
thumb
>>> thumb.set(RDFS.label, Literal("thumbnail"))
>>> print(thumb.label())
thumbnail
>>> thumb.remove(RDFS.label)
>>> list(thumb.objects(RDFS.label))
[]
```

Schema Example

With this artificial schema data:

```
>>> graph = Graph().parse(format='n3', data='''
... @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
... @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
... @prefix owl: <http://www.w3.org/2002/07/owl#> .
... @prefix v: <http://example.org/def/v#> .
...
... v:Artifact a owl:Class .
...
... v:Document a owl:Class;
...     rdfs:subClassOf v:Artifact .
...
... v:Paper a owl:Class;
...     rdfs:subClassOf v:Document .
...
... v:Choice owl:oneOf (v:One v:Other) .
...
... v:Stuff a rdf:Seq; rdf:_1 v:One; rdf:_2 v:Other .
...
... ''')
```

From this class:

```
>>> artifact = Resource(graph, URIRef("http://example.org/def/v#Artifact"))
```

we can get at subclasses:

```
>>> subclasses = list(artifact.transitive_subjects(RDFS.subClassOf))
>>> [c.qname() for c in subclasses]
[u'v:Artifact', u'v:Document', u'v:Paper']
```

and superclasses from the last subclass:

```
>>> [c.qname() for c in subclasses[-1].transitive_objects(RDFS.subClassOf)]
[u'v:Paper', u'v:Document', u'v:Artifact']
```

Get items from the Choice:

```
>>> choice = Resource(graph, URIRef("http://example.org/def/v#Choice"))
>>> [it.qname() for it in choice.value(OWL.oneOf).items()]
[u'v:One', u'v:Other']
```

And the sequence of Stuff:

```
>>> stuff = Resource(graph, URIRef("http://example.org/def/v#Stuff"))
>>> [it.qname() for it in stuff.seq()]
[u'v:One', u'v:Other']
```

On add, other resources are auto-unboxed:

```
>>> paper = Resource(graph, URIRef("http://example.org/def/v#Paper"))
>>> paper.add(RDFS.subClassOf, artifact)
>>> artifact in paper.objects(RDFS.subClassOf) # checks Resource instance
True
>>> (paper._identifier, RDFS.subClassOf, artifact._identifier) in graph
True
```

Technical Details

Comparison is based on graph and identifier:

```
>>> g1 = Graph()
>>> t1 = Resource(g1, URIRef("http://example.org/thing"))
>>> t2 = Resource(g1, URIRef("http://example.org/thing"))
>>> t3 = Resource(g1, URIRef("http://example.org/other"))
>>> t4 = Resource(Graph(), URIRef("http://example.org/other"))

>>> t1 is t2
False

>>> t1 == t2
True
>>> t1 != t2
False

>>> t1 == t3
False
>>> t1 != t3
True

>>> t3 != t4
True

>>> t3 < t1 and t1 > t3
True
>>> t1 >= t1 and t1 >= t3
True
>>> t1 <= t1 and t3 <= t1
```

(continues on next page)

(continued from previous page)

```
True

>>> t1 < t1 or t1 < t3 or t3 > t1 or t3 > t3
False
```

Hash is computed from graph and identifier:

```
>>> g1 = Graph()
>>> t1 = Resource(g1, URIRef("http://example.org/thing"))

>>> hash(t1) == hash(Resource(g1, URIRef("http://example.org/thing")))
True

>>> hash(t1) == hash(Resource(Graph(), t1.identifier))
False
>>> hash(t1) == hash(Resource(Graph(), URIRef("http://example.org/thing")))
False
```

The Resource class is suitable as a base class for mapper toolkits. For example, consider this utility for accessing RDF properties via qname-like attributes:

```
>>> class Item(Resource):
...
...     def __getattr__(self, p):
...         return list(self.objects(self._to_ref(*p.split('_', 1))))
...
...     def _to_ref(self, pfx, name):
...         return URIRef(self._graph.store.namespace(pfx) + name)
```

It works as follows:

```
>>> graph = Graph().parse(format='n3', data='''
... @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
... @prefix foaf: <http://xmlns.com/foaf/0.1/> .
...
... @base <http://example.org/> .
... </person/some1#self>
...     foaf:name "Some Body";
...     foaf:depiction </images/person/some1.jpg> .
... </images/person/some1.jpg> rdfs:comment "Just an image"@en .
... ''')

>>> person = Item(graph, URIRef("http://example.org/person/some1#self"))

>>> print(person.foaf_name[0])
Some Body
```

The mechanism for wrapping references as resources cooperates with subclasses. Therefore, accessing referenced resources automatically creates new `Item` objects:

```
>>> isinstance(person.foaf_depiction[0], Item)
True

>>> print(person.foaf_depiction[0].rdfs_comment[0])
Just an image
```

class rdflib.resource.**Resource**(*graph*, *subject*)

Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.resource', '__init__': <function Reso

__eq__(*other*)
Return self==value.

__ge__(*other*)
Return self>=value.

__getitem__(*item*)

__gt__(*other*)
Return self>value.

__hash__()
Return hash(self).

__init__(*graph*, *subject*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()

__le__(*other*)
Return self<=value.

__lt__(*other*)
Return self<value.

__module__ = 'rdflib.resource'

__ne__(*other*)
Return self!=value.

__repr__()
Return repr(self).

__setitem__(*item*, *value*)

__str__()
Return str(self).

__unicode__()

__weakref__
list of weak references to the object (if defined)

add(*p*, *o*)

comment()

property graph

property identifier

items()

label()

objects(*predicate=None*)

predicate_objects()

predicates(*o=None*)

qname()

remove(*p*, *o=None*)

seq()

set(*p*, *o*)

subject_objects()

subject_predicates()

subjects(*predicate=None*)

transitive_objects(*predicate*, *remember=None*)

transitive_subjects(*predicate*, *remember=None*)

value(*p=rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#value')*, *o=None*, *default=None*, *any=True*)

rdflib.serializer module

Serializer plugin interface.

This module is useful for those wanting to write a serializer that can plugin to rdflib. If you are wanting to invoke a serializer you likely want to do so through the Graph class serialize method.

TODO: info for how to write a serializer that can plugin to rdflib. See also rdflib.plugin

class rdflib.serializer.**Serializer**(*store*)
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.serializer', '__init__': <function Se

__init__(*store*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.serializer'

__weakref__
list of weak references to the object (if defined)

relativize(*uri*)

serialize(*stream*, *base=None*, *encoding=None*, ***args*)
Abstract method

rdflib.store module

class rdflib.store.**StoreCreatedEvent**(***kw*)
Bases: *rdflib.events.Event*

This event is fired when the Store is created, it has the following attribute:

- `configuration`: string used to create the store

__module__ = 'rdflib.store'

class rdflib.store.**TripleAddedEvent**(***kw*)
Bases: *rdflib.events.Event*

This event is fired when a triple is added, it has the following attributes:

- the `triple` added to the graph
- the `context` of the triple, if any
- the `graph` to which the triple was added

__module__ = 'rdflib.store'

class rdflib.store.**TripleRemovedEvent**(***kw*)
Bases: *rdflib.events.Event*

This event is fired when a triple is removed, it has the following attributes:

- the `triple` removed from the graph
- the `context` of the triple, if any
- the `graph` from which the triple was removed

__module__ = 'rdflib.store'

class rdflib.store.**NodePickler**
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.store', '__init__': <function NodePic

__getstate__()

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.store'

__setstate__(*state*)

__weakref__
list of weak references to the object (if defined)

dumps(*obj*, *protocol=None*, *bin=None*)

loads(*s*)

register(*object*, *id*)

class rdflib.store.**Store**(*configuration=None*, *identifier=None*)
Bases: object

__dict__ = mappingproxy({'__module__': 'rdflib.store', 'context_aware': False, 'form

__init__(*configuration=None*, *identifier=None*)
identifier: URIRef of the Store. Defaults to CWD configuration: string containing infomation open can use to connect to datastore.

__len__(*context=None*)
Number of statements in the store. This should only account for non- quoted (asserted) statements if the context is not specified, otherwise it should return the number of statements in the formula or context given.

Parameters **context** – a graph instance to query or None

__module__ = 'rdflib.store'

__weakref__
list of weak references to the object (if defined)

add(*triple*, *context*, *quoted=False*)
Adds the given statement to a specific context or to the model. The quoted argument is interpreted by formula-aware stores to indicate this statement is quoted/hypothetical It should be an error to not specify a context and have the quoted argument be True. It should also be an error for the quoted argument to be True when the store is not formula-aware.

addN(*quads*)
Adds each item in the list of statements to a specific context. The quoted argument is interpreted by formula-aware stores to indicate this statement is quoted/hypothetical. Note that the default implementation is a redirect to add

add_graph(*graph*)
Add a graph to the store, no effect if the graph already exists. :param graph: a Graph instance

bind(*prefix*, *namespace*)

close(*commit_pending_transaction=False*)
This closes the database connection. The commit_pending_transaction parameter specifies whether to commit all pending transactions before closing (if the store is transactional).

commit()

context_aware = False

contexts(*triple=None*)
Generator over all contexts in the graph. If triple is specified, a generator over all contexts the triple is in.

if store is graph_aware, may also return empty contexts

Returns a generator over Nodes

create(*configuration*)

destroy(*configuration*)
This destroys the instance of the store identified by the configuration string.

formula_aware = False

gc()
Allows the store to perform any needed garbage collection

graph_aware = False

namespace(*prefix*)

namespaces()

property node_pickler

open(*configuration*, *create=False*)
Opens the store specified by the configuration string. If create is True a store will be created if it does not already exist. If create is False and a store does not already exist an exception is raised. An exception is also raised if a store exists, but there is insufficient permissions to open the store. This should return one of: VALID_STORE, CORRUPTED_STORE, or NO_STORE

prefix(*namespace*)

query(*query*, *initNs*, *initBindings*, *queryGraph*, ***kwargs*)
If stores provide their own SPARQL implementation, override this.

queryGraph is None, a URIRef or '__UNION__' If None the graph is specified in the query-string/object If URIRef it specifies the graph to query, If '__UNION__' the union of all named graphs should be queried (This is used by ConjunctiveGraphs Values other than None obviously only makes sense for context-aware stores.)

remove(*triple*, *context=None*)
Remove the set of triples matching the pattern from the store

remove_graph(*graph*)
Remove a graph from the store, this shoud also remove all triples in the graph

Parameters **graphid** – a Graph instance

rollback()

transaction_aware = False

triples(*triple_pattern*, *context=None*)
A generator over all the triples matching the pattern. Pattern can include any objects for used for comparing against nodes in the store, for example, REGEXTerm, URIRef, Literal, BNode, Variable, Graph, QuotedGraph, Date? DateRange?

Parameters **context** – A conjunctive query can be indicated by either providing a value of None, or a specific context can be queries by passing a Graph instance (if store is context aware).

triples_choices(*triple*, *context=None*)
A variant of triples that can take a list of terms instead of a single term in any slot. Stores can implement this to optimize the response time from the default 'fallback' implementation, which will iterate over each term in the list and dispatch to triples

update(*update*, *initNs*, *initBindings*, *queryGraph*, ***kwargs*)
If stores provide their own (SPARQL) Update implementation, override this.

queryGraph is None, a URIRef or '__UNION__' If None the graph is specified in the query-string/object If URIRef it specifies the graph to query, If '__UNION__' the union of all named graphs should be queried (This is used by ConjunctiveGraphs Values other than None obviously only makes sense for context-aware stores.)

rdflib.term module

This module defines the different types of terms. Terms are the kinds of objects that can appear in a quoted/asserted triple. This includes those that are core to RDF:

- *Blank Nodes*
- *URI References*
- *Literals* (which consist of a literal value,datatype and language tag)

Those that extend the RDF model into N3:

- *Formulae*
- *Universal Quantifications (Variables)*

And those that are primarily for matching against 'Nodes' in the underlying Graph:

- REGEX Expressions
- Date Ranges
- Numerical Ranges

rdflib.term.**bind**(*datatype*, *pythontype*, *constructor=None*, *lexicalizer=None*, *datatype_specific=False*)
register a new datatype<->pythontype binding

Parameters

- **constructor** – an optional function for converting lexical forms into a Python instances, if not given the pythontype is used directly
- **lexicalizer** – an optional function for converting python objects to lexical form, if not given object.__str__ is used

- **datatype_specific** – makes the lexicalizer function be accessible from the pair (pythontype, datatype) if set to True or from the pythontype otherwise. False by default

class rdflib.term.**Node**

Bases: object

A Node in the Graph.

__module__ = 'rdflib.term'

__slots__ = ()

class rdflib.term.**Identifier**

Bases: *rdflib.term.Node*, str

See http://www.w3.org/2002/07/rdf-identifer-terminology/ regarding choice of terminology.

__eq__(*other*)

Equality for Nodes.

```
>>> BNode("foo")==None
False
>>> BNode("foo")==URIRef("foo")
False
>>> URIRef("foo")==BNode("foo")
False
>>> BNode("foo")!=URIRef("foo")
True
>>> URIRef("foo")!=BNode("foo")
True
>>> Variable('a')!=URIRef('a')
True
>>> Variable('a')!=Variable('a')
False
```

__ge__(*other*)

Return self>=value.

__gt__(*other*)

This implements ordering for Nodes,

This tries to implement this: http://www.w3.org/TR/sparql11-query/#modOrderBy

Variables are not included in the SPARQL list, but they are greater than BNodes and smaller than everything else

__hash__()

Return hash(self).

__le__(*other*)

Return self<=value.

__lt__(*other*)

Return self<value.

__module__ = 'rdflib.term'

__ne__(*other*)

Return self!=value.

static __new__(*cls*, *value*)

Create and return a new object. See help(type) for accurate signature.

__slots__ = ()

eq(*other*)
A "semantic"/interpreted equality function, by default, same as __eq__

neq(*other*)
A "semantic"/interpreted not equal function, by default, same as __ne__

class rdflib.term.**URIRef**
Bases: *rdflib.term.Identifier*

RDF URI Reference: http://www.w3.org/TR/rdf-concepts/#section-Graph-URIref

__add__(*other*)
Return self+value.

__getnewargs__()

__invert__()
inverse path

__mod__(*other*)
Return self%value.

__module__ = 'rdflib.term'

__mul__(*mul*)
cardinality path

__neg__()
negated path

static __new__(*cls*, *value*, *base=None*)
Create and return a new object. See help(type) for accurate signature.

__or__(*other*)
alternative path

__radd__(*other*)

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__slots__ = ()

__truediv__(*other*)
sequence path

de_skolemize()
Create a Blank Node from a skolem URI, in accordance with http://www.w3.org/TR/rdf11-concepts/#section-skolemization. This function accepts only rdflib type skolemization, to provide a round-tripping within the system.

New in version 4.0.

defrag()

n3(*namespace_manager=None*)
This will do a limited check for valid URIs, essentially just making sure that the string includes no illegal characters (<, >, ", {, }, |, \, `, ^)

Parameters **namespace_manager** – if not None, will be used to make up a prefixed name

toPython()

class rdflib.term.**BNode**

Bases: *rdflib.term.Identifier*

Blank Node: http://www.w3.org/TR/rdf-concepts/#section-blank-nodes

__getnewargs__()

__module__ = 'rdflib.term'

static __new__(*cls*, *value=None*, *_sn_gen=<function _serial_number_generator.<locals>._generator>*, *_prefix='N'*)

only store implementations should pass in a value

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__slots__ = ()

n3(*namespace_manager=None*)

skolemize(*authority=None*, *basepath=None*)

Create a URIRef "skolem" representation of the BNode, in accordance with http://www.w3.org/TR/rdf11-concepts/#section-skolemization

New in version 4.0.

toPython()

class rdflib.term.**Literal**

Bases: *rdflib.term.Identifier*

RDF Literal: http://www.w3.org/TR/rdf-concepts/#section-Graph-Literal

The lexical value of the literal is the unicode object The interpreted, datatyped value is available from .value

Language tags must be valid according to :rfc:5646

For valid XSD datatypes, the lexical form is optionally normalized at construction time. Default behaviour is set by rdflib.NORMALIZE_LITERALS and can be overridden by the normalize parameter to __new__

Equality and hashing of Literals are done based on the lexical form, i.e.:

```
>>> from rdflib.namespace import XSD
```

```
>>> Literal('01')!=Literal('1') # clear - strings differ
True
```

but with data-type they get normalized:

```
>>> Literal('01', datatype=XSD.integer)!=Literal('1', datatype=XSD.integer)
False
```

unless disabled:

```
>>> Literal('01', datatype=XSD.integer, normalize=False)!=Literal('1',␣
↪datatype=XSD.integer)
True
```

Value based comparison is possible:

```
>>> Literal('01', datatype=XSD.integer).eq(Literal('1', datatype=XSD.float))
True
```

The eq method also provides limited support for basic python types:

```
>>> Literal(1).eq(1) # fine - int compatible with xsd:integer
True
>>> Literal('a').eq('b') # fine - str compatible with plain-lit
False
>>> Literal('a', datatype=XSD.string).eq('a') # fine - str compatible with␣
↪xsd:string
True
>>> Literal('a').eq(1) # not fine, int incompatible with plain-lit
NotImplemented
```

Greater-than/less-than ordering comparisons are also done in value space, when compatible datatypes are used. Incompatible datatypes are ordered by DT, or by lang-tag. For other nodes the ordering is None < BNode < URIRef < Literal

Any comparison with non-rdflib Node are "NotImplemented" In PY3 this is an error.

```
>>> from rdflib import Literal, XSD
>>> lit2006 = Literal('2006-01-01',datatype=XSD.date)
>>> lit2006.toPython()
datetime.date(2006, 1, 1)
>>> lit2006 < Literal('2007-01-01',datatype=XSD.date)
True
>>> Literal(datetime.utcnow()).datatype
rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#dateTime')
>>> Literal(1) > Literal(2) # by value
False
>>> Literal(1) > Literal(2.0) # by value
False
>>> Literal('1') > Literal(1) # by DT
True
>>> Literal('1') < Literal('1') # by lexical form
False
>>> Literal('a', lang='en') > Literal('a', lang='fr') # by lang-tag
False
>>> Literal(1) > URIRef('foo') # by node-type
True
```

The > < operators will eat this NotImplemented and throw a TypeError (py3k):

```
>>> Literal(1).__gt__(2.0)
NotImplemented
```

__abs__()

```
>>> abs(Literal(-1))
rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

```
>>> from rdflib.namespace import XSD
>>> abs( Literal("-1", datatype=XSD.integer))
rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

(continues on next page)

(continued from previous page)

```
>>> abs(Literal("1"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Not a number; rdflib.term.Literal(u'1')
```

__add__(*val*)

```
>>> Literal(1) + 1
rdflib.term.Literal(u'2', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
>>> Literal("1") + "1"
rdflib.term.Literal(u'11')
```

__bool__()
Is the Literal "True" This is used for if statements, bool(literal), etc.

__eq__(*other*)
Literals are only equal to other literals.

"Two literals are equal if and only if all of the following hold: * The strings of the two lexical forms compare equal, character by character. * Either both or neither have language tags. * The language tags, if any, compare equal. * Either both or neither have datatype URIs. * The two datatype URIs, if any, compare equal, character by character." – 6.5.1 Literal Equality (RDF: Concepts and Abstract Syntax)

```
>>> Literal("1", datatype=URIRef("foo")) == Literal("1", datatype=URIRef("foo
↪"))
True
>>> Literal("1", datatype=URIRef("foo")) == Literal("1", datatype=URIRef("foo2
↪"))
False
```

```
>>> Literal("1", datatype=URIRef("foo")) == Literal("2", datatype=URIRef("foo
↪"))
False
>>> Literal("1", datatype=URIRef("foo")) == "asdf"
False
>>> from rdflib import XSD
>>> Literal('2007-01-01', datatype=XSD.date) == Literal('2007-01-01',␣
↪datatype=XSD.date)
True
>>> Literal('2007-01-01', datatype=XSD.date) == date(2007, 1, 1)
False
>>> Literal("one", lang="en") == Literal("one", lang="en")
True
>>> Literal("hast", lang='en') == Literal("hast", lang='de')
False
>>> Literal("1", datatype=XSD.integer) == Literal(1)
True
>>> Literal("1", datatype=XSD.integer) == Literal("01", datatype=XSD.integer)
True
```

__ge__(*other*)
Return self>=value.

__getstate__()

__gt__(*other*)

This implements ordering for Literals, the other comparison methods delegate here

This tries to implement this: http://www.w3.org/TR/sparql11-query/#modOrderBy

In short, Literals with compatible data-types are ordered in value space, i.e. >>> from rdflib import XSD

```
>>> Literal(1) > Literal(2) # int/int
False
>>> Literal(2.0) > Literal(1) # double/int
True
>>> from decimal import Decimal
>>> Literal(Decimal("3.3")) > Literal(2.0) # decimal/double
True
>>> Literal(Decimal("3.3")) < Literal(4.0) # decimal/double
True
>>> Literal('b') > Literal('a') # plain lit/plain lit
True
>>> Literal('b') > Literal('a', datatype=XSD.string) # plain lit/xsd:str
True
```

Incompatible datatype mismatches ordered by DT

```
>>> Literal(1) > Literal("2") # int>string
False
```

Langtagged literals by lang tag >>> Literal("a", lang="en") > Literal("a", lang="fr") False

__hash__()

```
>>> from rdflib.namespace import XSD
>>> a = {Literal('1', datatype=XSD.integer):'one'}
>>> Literal('1', datatype=XSD.double) in a
False
```

"Called for the key object for dictionary operations, and by the built-in function hash(). Should return a 32-bit integer usable as a hash value for dictionary operations. The only required property is that objects which compare equal have the same hash value; it is advised to somehow mix together (e.g., using exclusive or) the hash values for the components of the object that also play a part in comparison of objects." – 3.4.1 Basic customization (Python)

"Two literals are equal if and only if all of the following hold: * The strings of the two lexical forms compare equal, character by character. * Either both or neither have language tags. * The language tags, if any, compare equal. * Either both or neither have datatype URIs. * The two datatype URIs, if any, compare equal, character by character." – 6.5.1 Literal Equality (RDF: Concepts and Abstract Syntax)

__invert__()

```
>>> ~(Literal(-1))
rdflib.term.Literal(u'0', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

```
>>> from rdflib.namespace import XSD
>>> ~( Literal("-1", datatype=XSD.integer))
rdflib.term.Literal(u'0', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

(continues on next page)

(continued from previous page)

Not working:

```
>>> ~(Literal("1"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Not a number; rdflib.term.Literal(u'1')
```

__le__(*other*)

```
>>> from rdflib.namespace import XSD
>>> Literal('2007-01-01T10:00:00', datatype=XSD.dateTime
...     ) <= Literal('2007-01-01T10:00:00', datatype=XSD.dateTime)
True
```

__lt__(*other*)
Return self<value.

__module__ = 'rdflib.term'

__neg__()

```
>>> (- Literal(1))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
>>> (- Literal(10.5))
rdflib.term.Literal(u'-10.5', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#double'))
>>> from rdflib.namespace import XSD
>>> (- Literal("1", datatype=XSD.integer))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

```
>>> (- Literal("1"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Not a number; rdflib.term.Literal(u'1')
>>>
```

static __new__(*cls*, *lexical_or_value*, *lang=None*, *datatype=None*, *normalize=None*)
Create and return a new object. See help(type) for accurate signature.

__pos__()

```
>>> (+ Literal(1))
rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
>>> (+ Literal(-1))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
>>> from rdflib.namespace import XSD
>>> (+ Literal("-1", datatype=XSD.integer))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

```
>>> (+ Literal("1"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Not a number; rdflib.term.Literal(u'1')
```

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__setstate__(*arg*)

__slots__ = ('_language', '_datatype', '_value')

property datatype

eq(*other*)
Compare the value of this literal with something else

Either, with the value of another literal comparisons are then done in literal "value space", and according to the rules of XSD subtype-substitution/type-promotion

OR, with a python object:

basestring objects can be compared with plain-literals, or those with datatype xsd:string

bool objects with xsd:boolean

a int, long or float with numeric xsd types

isodate date,time,datetime objects with xsd:date,xsd:time or xsd:datetime

Any other operations returns NotImplemented

property language

n3(*namespace_manager=None*)
Returns a representation in the N3 format.

Examples:

```
>>> Literal("foo").n3()
u'"foo"'
```

Strings with newlines or triple-quotes:

```
>>> Literal("foo\nbar").n3()
u'"""foo\nbar"""'

>>> Literal("'''\'").n3()
u'"\'\'\'"'

>>> Literal('""""').n3()
u'"\\"\\"\\""'
```

Language:

```
>>> Literal("hello", lang="en").n3()
u'"hello"@en'
```

Datatypes:

```
>>> Literal(1).n3()
u'"1"^^<http://www.w3.org/2001/XMLSchema#integer>'

>>> Literal(1.0).n3()
u'"1.0"^^<http://www.w3.org/2001/XMLSchema#double>'

>>> Literal(True).n3()
u'"true"^^<http://www.w3.org/2001/XMLSchema#boolean>'
```

Datatype and language isn't allowed (datatype takes precedence):

```
>>> Literal(1, lang="en").n3()
u'"1"^^<http://www.w3.org/2001/XMLSchema#integer>'
```

Custom datatype:

```
>>> footype = URIRef("http://example.org/ns#foo")
>>> Literal("1", datatype=footype).n3()
u'"1"^^<http://example.org/ns#foo>'
```

Passing a namespace-manager will use it to abbreviate datatype URIs:

```
>>> from rdflib import Graph
>>> Literal(1).n3(Graph().namespace_manager)
u'"1"^^xsd:integer'
```

neq(*other*)
A "semantic"/interpreted not equal function, by default, same as __ne__

normalize()
Returns a new literal with a normalised lexical representation of this literal >>> from rdflib import XSD >>> Literal("01", datatype=XSD.integer, normalize=False).normalize() rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#integer'))

Illegal lexical forms for the datatype given are simply passed on >>> Literal("a", datatype=XSD.integer, normalize=False) rdflib.term.Literal(u'a', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#integer'))

toPython()
Returns an appropriate python datatype derived from this RDF Literal

property value

class rdflib.term.**Variable**
Bases: *rdflib.term.Identifier*

A Variable - this is used for querying, or in Formula aware graphs, where Variables can stored in the graph

__module__ = 'rdflib.term'

static __new__(*cls*, *value*)
Create and return a new object. See help(type) for accurate signature.

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__slots__ = ()

n3(*namespace_manager=None*)

toPython()

class rdflib.term.**Statement**
Bases: *rdflib.term.Node*, tuple

__dict__ = mappingproxy({'__module__': 'rdflib.term', '__new__': <staticmethod objec

__module__ = 'rdflib.term'

static __new__(*cls*, *triple*, *context*)
Create and return a new object. See help(type) for accurate signature.

__reduce__()
Helper for pickle.

toPython()

rdflib.util module

Some utility functions.

Miscellaneous utilities

- list2set
- first
- uniq
- more_than

Term characterisation and generation

- to_term
- from_n3

Date/time utilities

- date_time
- parse_date_time

Statement and component type checkers

- check_context
- check_subject
- check_predicate
- check_object
- check_statement
- check_pattern

rdflib.util.**list2set**(*seq*)
Return a new list without duplicates. Preserves the order, unlike set(seq)

rdflib.util.**first**(*seq*)
return the first element in a python sequence for graphs, use graph.value instead

rdflib.util.**uniq**(*sequence*, *strip=0*)
removes duplicate strings from the sequence.

rdflib.util.**more_than**(*sequence*, *number*)
Returns 1 if sequence has more items than number and 0 if not.

rdflib.util.**to_term**(*s*, *default=None*)
Creates and returns an Identifier of type corresponding to the pattern of the given positional argument string `s`:

'' returns the `default` keyword argument value or `None`

'<s>' returns `URIRef(s)` (i.e. without angle brackets)

'"s"' returns `Literal(s)` (i.e. without doublequotes)

'_s' returns `BNode(s)` (i.e. without leading underscore)

rdflib.util.**from_n3**(*s*, *default=None*, *backend=None*, *nsm=None*)
Creates the Identifier corresponding to the given n3 string.

```
>>> from_n3('<http://ex.com/foo>') == URIRef('http://ex.com/foo')
True
>>> from_n3('"foo"@de') == Literal('foo', lang='de')
True
>>> from_n3('"""multi\nline\nstring"""@en') == Literal(
...     'multi\nline\nstring', lang='en')
True
>>> from_n3('42') == Literal(42)
True
>>> from_n3(Literal(42).n3()) == Literal(42)
True
>>> from_n3('"42"^^xsd:integer') == Literal(42)
True
>>> from rdflib import RDFS
>>> from_n3('rdfs:label') == RDFS['label']
True
>>> nsm = NamespaceManager(Graph())
>>> nsm.bind('dbpedia', 'http://dbpedia.org/resource/')
>>> berlin = URIRef('http://dbpedia.org/resource/Berlin')
>>> from_n3('dbpedia:Berlin', nsm=nsm) == berlin
True
```

rdflib.util.**date_time**(*t=None*, *local_time_zone=False*)
http://www.w3.org/TR/NOTE-datetime ex: 1997-07-16T19:20:30Z

```
>>> date_time(1126482850)
'2005-09-11T23:54:10Z'
```

@@ this will change depending on where it is run #>>> date_time(1126482850, local_time_zone=True) #'2005-09-11T19:54:10-04:00'

```
>>> date_time(1)
'1970-01-01T00:00:01Z'
```

```
>>> date_time(0)
'1970-01-01T00:00:00Z'
```

rdflib.util.**parse_date_time**(*val*)
always returns seconds in UTC

tests are written like this to make any errors easier to understand >>> parse_date_time('2005-09-11T23:54:10Z') - 1126482850.0 0.0

```
>>> parse_date_time('2005-09-11T16:54:10-07:00') - 1126482850.0
0.0
```

```
>>> parse_date_time('1970-01-01T00:00:01Z') - 1.0
0.0
```

```
>>> parse_date_time('1970-01-01T00:00:00Z') - 0.0
0.0
>>> parse_date_time("2005-09-05T10:42:00") - 1125916920.0
0.0
```

rdflib.util.**check_context**(*c*)

rdflib.util.**check_subject**(*s*)
Test that s is a valid subject identifier.

rdflib.util.**check_predicate**(*p*)
Test that p is a valid predicate identifier.

rdflib.util.**check_object**(*o*)
Test that o is a valid object identifier.

rdflib.util.**check_statement**(*triple*)

rdflib.util.**check_pattern**(*triple*)

rdflib.util.**guess_format**(*fpath*, *fmap=None*)
Guess RDF serialization based on file suffix. Uses `SUFFIX_FORMAT_MAP` unless `fmap` is provided. Examples:

```
>>> guess_format('path/to/file.rdf')
'xml'
>>> guess_format('path/to/file.owl')
'xml'
>>> guess_format('path/to/file.ttl')
'turtle'
>>> guess_format('path/to/file.xhtml')
'rdfa'
>>> guess_format('path/to/file.svg')
'rdfa'
>>> guess_format('path/to/file.xhtml', {'xhtml': 'grddl'})
'grddl'
```

This also works with just the suffixes, with or without leading dot, and regardless of letter case:

```
>>> guess_format('.rdf')
'xml'
>>> guess_format('rdf')
'xml'
>>> guess_format('RDF')
'xml'
```

rdflib.util.**find_roots**(*graph*, *prop*, *roots=None*)
Find the roots in some sort of transitive hierarchy.

find_roots(graph, rdflib.RDFS.subClassOf) will return a set of all roots of the sub-class hierarchy

Assumes triple of the form (child, prop, parent), i.e. the direction of RDFS.subClassOf or SKOS.broader

rdflib.util.**get_tree**(*graph*, *root*, *prop*, *mapper=<function <lambda>>*, *sortkey=None*, *done=None*, *dir='down'*)

Return a nested list/tuple structure representing the tree built by the transitive property given, starting from the root given

i.e.

get_tree(graph, rdflib.URIRef("http://xmlns.com/foaf/0.1/Person"), rdflib.RDFS.subClassOf)

will return the structure for the subClassTree below person.

dir='down' assumes triple of the form (child, prop, parent), i.e. the direction of RDFS.subClassOf or SKOS.broader Any other dir traverses in the other direction

rdflib.void module

rdflib.void.**generateVoID**(*g*, *dataset=None*, *res=None*, *distinctForPartitions=True*)

Returns a new graph with a VoID description of the passed dataset

For more info on Vocabulary of Interlinked Datasets (VoID), see: http://vocab.deri.ie/void

This only makes two passes through the triples (once to detect the types of things)

The tradeoff is that lots of temporary structures are built up in memory meaning lots of memory may be consumed :) I imagine at least a few copies of your original graph.

the distinctForPartitions parameter controls whether distinctSubjects/objects are tracked for each class/propertyPartition this requires more memory again

Module contents

A pure Python package providing the core RDF constructs.

The packages is intended to provide the core RDF types and interfaces for working with RDF. The package defines a plugin interface for parsers, stores, and serializers that other packages can use to implement parsers, stores, and serializers that will plug into the rdflib package.

The primary interface *rdflib* exposes to work with RDF is *rdflib.graph.Graph*.

A tiny example:

```
>>> from rdflib import Graph, URIRef, Literal
```

```
>>> g = Graph()
>>> result = g.parse("http://www.w3.org/2000/10/swap/test/meet/blue.rdf")
```

```
>>> print("graph has %s statements." % len(g))
graph has 4 statements.
>>>
>>> for s, p, o in g:
...     if (s, p, o) not in g:
...         raise Exception("It better be!")
```

```
>>> s = g.serialize(format='nt')
>>>
>>> sorted(g) == [
...  (URIRef(u'http://meetings.example.com/cal#m1'),
...   URIRef(u'http://www.example.org/meeting_organization#homePage'),
```

(continues on next page)

(continued from previous page)

```
...    URIRef(u'http://meetings.example.com/m1/hp')),
...   (URIRef(u'http://www.example.org/people#fred'),
...    URIRef(u'http://www.example.org/meeting_organization#attending'),
...    URIRef(u'http://meetings.example.com/cal#m1')),
...   (URIRef(u'http://www.example.org/people#fred'),
...    URIRef(u'http://www.example.org/personal_details#GivenName'),
...    Literal(u'Fred')),
...   (URIRef(u'http://www.example.org/people#fred'),
...    URIRef(u'http://www.example.org/personal_details#hasEmail'),
...    URIRef(u'mailto:fred@example.com'))
... ]
True
```

class rdflib.**URIRef**

Bases: *rdflib.term.Identifier*

RDF URI Reference: http://www.w3.org/TR/rdf-concepts/#section-Graph-URIref

__add__(*other*)
Return self+value.

__getnewargs__()

__invert__()
inverse path

__mod__(*other*)
Return self%value.

__module__ = 'rdflib.term'

__mul__(*mul*)
cardinality path

__neg__()
negated path

static __new__(*cls*, *value*, *base=None*)
Create and return a new object. See help(type) for accurate signature.

__or__(*other*)
alternative path

__radd__(*other*)

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__slots__ = ()

__truediv__(*other*)
sequence path

de_skolemize()
Create a Blank Node from a skolem URI, in accordance with http://www.w3.org/TR/rdf11-concepts/#section-skolemization. This function accepts only rdflib type skolemization, to provide a round-tripping within the system.

New in version 4.0.

defrag()

n3(*namespace_manager=None*)
This will do a limited check for valid URIs, essentially just making sure that the string includes no illegal characters (<, >, ", {, }, |, \, `, ^)

Parameters **namespace_manager** – if not None, will be used to make up a prefixed name

toPython()

class rdflib.**BNode**
Bases: *rdflib.term.Identifier*

Blank Node: http://www.w3.org/TR/rdf-concepts/#section-blank-nodes

__getnewargs__()

__module__ = 'rdflib.term'

static __new__(*cls*, *value=None*, *_sn_gen=<function _serial_number_generator.<locals>._generator>*, *_prefix='N'*)
only store implementations should pass in a value

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__slots__ = ()

n3(*namespace_manager=None*)

skolemize(*authority=None*, *basepath=None*)
Create a URIRef "skolem" representation of the BNode, in accordance with http://www.w3.org/TR/rdf11-concepts/#section-skolemization

New in version 4.0.

toPython()

class rdflib.**Literal**
Bases: *rdflib.term.Identifier*

RDF Literal: http://www.w3.org/TR/rdf-concepts/#section-Graph-Literal

The lexical value of the literal is the unicode object The interpreted, datatyped value is available from .value

Language tags must be valid according to :rfc:5646

For valid XSD datatypes, the lexical form is optionally normalized at construction time. Default behaviour is set by rdflib.NORMALIZE_LITERALS and can be overridden by the normalize parameter to __new__

Equality and hashing of Literals are done based on the lexical form, i.e.:

```
>>> from rdflib.namespace import XSD
```

```
>>> Literal('01')!=Literal('1') # clear - strings differ
True
```

but with data-type they get normalized:

```
>>> Literal('01', datatype=XSD.integer)!=Literal('1', datatype=XSD.integer)
False
```

unless disabled:

```
>>> Literal('01', datatype=XSD.integer, normalize=False)!=Literal('1',␣
↪datatype=XSD.integer)
True
```

Value based comparison is possible:

```
>>> Literal('01', datatype=XSD.integer).eq(Literal('1', datatype=XSD.float))
True
```

The eq method also provides limited support for basic python types:

```
>>> Literal(1).eq(1) # fine - int compatible with xsd:integer
True
>>> Literal('a').eq('b') # fine - str compatible with plain-lit
False
>>> Literal('a', datatype=XSD.string).eq('a') # fine - str compatible with␣
↪xsd:string
True
>>> Literal('a').eq(1) # not fine, int incompatible with plain-lit
NotImplemented
```

Greater-than/less-than ordering comparisons are also done in value space, when compatible datatypes are used. Incompatible datatypes are ordered by DT, or by lang-tag. For other nodes the ordering is None < BNode < URIRef < Literal

Any comparison with non-rdflib Node are "NotImplemented" In PY3 this is an error.

```
>>> from rdflib import Literal, XSD
>>> lit2006 = Literal('2006-01-01',datatype=XSD.date)
>>> lit2006.toPython()
datetime.date(2006, 1, 1)
>>> lit2006 < Literal('2007-01-01',datatype=XSD.date)
True
>>> Literal(datetime.utcnow()).datatype
rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#dateTime')
>>> Literal(1) > Literal(2) # by value
False
>>> Literal(1) > Literal(2.0) # by value
False
>>> Literal('1') > Literal(1) # by DT
True
>>> Literal('1') < Literal('1') # by lexical form
False
>>> Literal('a', lang='en') > Literal('a', lang='fr') # by lang-tag
False
>>> Literal(1) > URIRef('foo') # by node-type
True
```

The > < operators will eat this NotImplemented and throw a TypeError (py3k):

```
>>> Literal(1).__gt__(2.0)
NotImplemented
```

__abs__()

```
>>> abs(Literal(-1))
rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

```
>>> from rdflib.namespace import XSD
>>> abs( Literal("-1", datatype=XSD.integer))
rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

```
>>> abs(Literal("1"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Not a number; rdflib.term.Literal(u'1')
```

__add__(*val*)

```
>>> Literal(1) + 1
rdflib.term.Literal(u'2', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
>>> Literal("1") + "1"
rdflib.term.Literal(u'11')
```

__bool__()
Is the Literal "True" This is used for if statements, bool(literal), etc.

__eq__(*other*)
Literals are only equal to other literals.

"Two literals are equal if and only if all of the following hold: * The strings of the two lexical forms compare equal, character by character. * Either both or neither have language tags. * The language tags, if any, compare equal. * Either both or neither have datatype URIs. * The two datatype URIs, if any, compare equal, character by character." – 6.5.1 Literal Equality (RDF: Concepts and Abstract Syntax)

```
>>> Literal("1", datatype=URIRef("foo")) == Literal("1", datatype=URIRef("foo
↪"))
True
>>> Literal("1", datatype=URIRef("foo")) == Literal("1", datatype=URIRef("foo2
↪"))
False
```

```
>>> Literal("1", datatype=URIRef("foo")) == Literal("2", datatype=URIRef("foo
↪"))
False
>>> Literal("1", datatype=URIRef("foo")) == "asdf"
False
>>> from rdflib import XSD
>>> Literal('2007-01-01', datatype=XSD.date) == Literal('2007-01-01',␣
↪datatype=XSD.date)
True
>>> Literal('2007-01-01', datatype=XSD.date) == date(2007, 1, 1)
False
>>> Literal("one", lang="en") == Literal("one", lang="en")
True
>>> Literal("hast", lang='en') == Literal("hast", lang='de')
False
```

(continues on next page)

(continued from previous page)

```
>>> Literal("1", datatype=XSD.integer) == Literal(1)
True
>>> Literal("1", datatype=XSD.integer) == Literal("01", datatype=XSD.integer)
True
```

__ge__(*other*)
Return self>=value.

__getstate__()

__gt__(*other*)
This implements ordering for Literals, the other comparison methods delegate here

This tries to implement this: http://www.w3.org/TR/sparql11-query/#modOrderBy

In short, Literals with compatible data-types are ordered in value space, i.e. >>> from rdflib import XSD

```
>>> Literal(1) > Literal(2) # int/int
False
>>> Literal(2.0) > Literal(1) # double/int
True
>>> from decimal import Decimal
>>> Literal(Decimal("3.3")) > Literal(2.0) # decimal/double
True
>>> Literal(Decimal("3.3")) < Literal(4.0) # decimal/double
True
>>> Literal('b') > Literal('a') # plain lit/plain lit
True
>>> Literal('b') > Literal('a', datatype=XSD.string) # plain lit/xsd:str
True
```

Incompatible datatype mismatches ordered by DT

```
>>> Literal(1) > Literal("2") # int>string
False
```

Langtagged literals by lang tag >>> Literal("a", lang="en") > Literal("a", lang="fr") False

__hash__()

```
>>> from rdflib.namespace import XSD
>>> a = {Literal('1', datatype=XSD.integer):'one'}
>>> Literal('1', datatype=XSD.double) in a
False
```

"Called for the key object for dictionary operations, and by the built-in function hash(). Should return a 32-bit integer usable as a hash value for dictionary operations. The only required property is that objects which compare equal have the same hash value; it is advised to somehow mix together (e.g., using exclusive or) the hash values for the components of the object that also play a part in comparison of objects." – 3.4.1 Basic customization (Python)

"Two literals are equal if and only if all of the following hold: * The strings of the two lexical forms compare equal, character by character. * Either both or neither have language tags. * The language tags, if any, compare equal. * Either both or neither have datatype URIs. * The two datatype URIs, if any, compare equal, character by character." – 6.5.1 Literal Equality (RDF: Concepts and Abstract Syntax)

__invert__()

```
>>> ~(Literal(-1))
rdflib.term.Literal(u'0', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

```
>>> from rdflib.namespace import XSD
>>> ~( Literal("-1", datatype=XSD.integer))
rdflib.term.Literal(u'0', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

Not working:

```
>>> ~(Literal("1"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Not a number; rdflib.term.Literal(u'1')
```

__le__(*other*)

```
>>> from rdflib.namespace import XSD
>>> Literal('2007-01-01T10:00:00', datatype=XSD.dateTime
...     ) <= Literal('2007-01-01T10:00:00', datatype=XSD.dateTime)
True
```

__lt__(*other*)
Return self<value.

__module__ = 'rdflib.term'

__neg__()

```
>>> (- Literal(1))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
>>> (- Literal(10.5))
rdflib.term.Literal(u'-10.5', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#double'))
>>> from rdflib.namespace import XSD
>>> (- Literal("1", datatype=XSD.integer))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

```
>>> (- Literal("1"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Not a number; rdflib.term.Literal(u'1')
>>>
```

static __new__(*cls*, *lexical_or_value*, *lang=None*, *datatype=None*, *normalize=None*)
Create and return a new object. See help(type) for accurate signature.

__pos__()

```
>>> (+ Literal(1))
rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

(continues on next page)

(continued from previous page)

```
>>> (+ Literal(-1))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
>>> from rdflib.namespace import XSD
>>> (+ Literal("-1", datatype=XSD.integer))
rdflib.term.Literal(u'-1', datatype=rdflib.term.URIRef(u'http://www.w3.org/
↪2001/XMLSchema#integer'))
```

```
>>> (+ Literal("1"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Not a number; rdflib.term.Literal(u'1')
```

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__setstate__(*arg*)

__slots__ = ('_language', '_datatype', '_value')

property datatype

eq(*other*)
Compare the value of this literal with something else

Either, with the value of another literal comparisons are then done in literal "value space", and according to the rules of XSD subtype-substitution/type-promotion

OR, with a python object:

basestring objects can be compared with plain-literals, or those with datatype xsd:string

bool objects with xsd:boolean

a int, long or float with numeric xsd types

isodate date,time,datetime objects with xsd:date,xsd:time or xsd:datetime

Any other operations returns NotImplemented

property language

n3(*namespace_manager=None*)
Returns a representation in the N3 format.

Examples:

```
>>> Literal("foo").n3()
u'"foo"'
```

Strings with newlines or triple-quotes:

```
>>> Literal("foo\nbar").n3()
u'"""foo\nbar"""'

>>> Literal("'''\'").n3()
u'"\'\'\'\'"'
```

(continues on next page)

(continued from previous page)

```
>>> Literal('"""').n3()
u'"\\"\\"\\""'
```

Language:

```
>>> Literal("hello", lang="en").n3()
u'"hello"@en'
```

Datatypes:

```
>>> Literal(1).n3()
u'"1"^^<http://www.w3.org/2001/XMLSchema#integer>'

>>> Literal(1.0).n3()
u'"1.0"^^<http://www.w3.org/2001/XMLSchema#double>'

>>> Literal(True).n3()
u'"true"^^<http://www.w3.org/2001/XMLSchema#boolean>'
```

Datatype and language isn't allowed (datatype takes precedence):

```
>>> Literal(1, lang="en").n3()
u'"1"^^<http://www.w3.org/2001/XMLSchema#integer>'
```

Custom datatype:

```
>>> footype = URIRef("http://example.org/ns#foo")
>>> Literal("1", datatype=footype).n3()
u'"1"^^<http://example.org/ns#foo>'
```

Passing a namespace-manager will use it to abbreviate datatype URIs:

```
>>> from rdflib import Graph
>>> Literal(1).n3(Graph().namespace_manager)
u'"1"^^xsd:integer'
```

neq(*other*)
A "semantic"/interpreted not equal function, by default, same as __ne__

normalize()
Returns a new literal with a normalised lexical representation of this literal >>> from rdflib import XSD >>> Literal("01", datatype=XSD.integer, normalize=False).normalize() rdflib.term.Literal(u'1', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#integer'))

Illegal lexical forms for the datatype given are simply passed on >>> Literal("a", datatype=XSD.integer, normalize=False) rdflib.term.Literal(u'a', datatype=rdflib.term.URIRef(u'http://www.w3.org/2001/XMLSchema#integer'))

toPython()
Returns an appropriate python datatype derived from this RDF Literal

property value

class rdflib.**Variable**
Bases: *rdflib.term.Identifier*

A Variable - this is used for querying, or in Formula aware graphs, where Variables can stored in the graph

__module__ = 'rdflib.term'

static __new__(*cls*, *value*)
Create and return a new object. See help(type) for accurate signature.

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__slots__ = ()

n3(*namespace_manager=None*)

toPython()

class rdflib.**Namespace**
Bases: str

Utility class for quickly generating URIRefs with a common prefix

```
>>> from rdflib import Namespace
>>> n = Namespace("http://example.org/")
>>> n.Person # as attribute
rdflib.term.URIRef(u'http://example.org/Person')
>>> n['first-name'] # as item - for things that are not valid python identifiers
rdflib.term.URIRef(u'http://example.org/first-name')
```

__dict__ = mappingproxy({'__module__': 'rdflib.namespace', '__doc__': '\n Utility cl

__getattr__(*name*)

__getitem__(*key*, *default=None*)
Return self[key].

__module__ = 'rdflib.namespace'

static __new__(*cls*, *value*)
Create and return a new object. See help(type) for accurate signature.

__repr__()
Return repr(self).

__weakref__
list of weak references to the object (if defined)

term(*name*)

property title
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

class rdflib.**Dataset**(*store='default'*, *default_union=False*, *default_graph_base=None*)
Bases: *rdflib.graph.ConjunctiveGraph*

RDF 1.1 Dataset. Small extension to the Conjunctive Graph: - the primary term is graphs in the datasets and not contexts with quads, so there is a separate method to set/retrieve a graph in a dataset and operate with graphs - graphs cannot be identified with blank nodes - added a method to directly add a single quad

Examples of usage:

```
>>> # Create a new Dataset
>>> ds = Dataset()
>>> # simple triples goes to default graph
>>> ds.add((URIRef("http://example.org/a"),
...    URIRef("http://www.example.org/b"),
...    Literal("foo")))
>>>
>>> # Create a graph in the dataset, if the graph name has already been
>>> # used, the corresponding graph will be returned
>>> # (ie, the Dataset keeps track of the constituent graphs)
>>> g = ds.graph(URIRef("http://www.example.com/gr"))
>>>
>>> # add triples to the new graph as usual
>>> g.add(
...     (URIRef("http://example.org/x"),
...     URIRef("http://example.org/y"),
...     Literal("bar")) )
>>> # alternatively: add a quad to the dataset -> goes to the graph
>>> ds.add(
...     (URIRef("http://example.org/x"),
...     URIRef("http://example.org/z"),
...     Literal("foo-bar"),g) )
>>>
>>> # querying triples return them all regardless of the graph
>>> for t in ds.triples((None,None,None)):
...     print(t)
(rdflib.term.URIRef("http://example.org/a"),
 rdflib.term.URIRef("http://www.example.org/b"),
 rdflib.term.Literal("foo"))
(rdflib.term.URIRef("http://example.org/x"),
 rdflib.term.URIRef("http://example.org/z"),
 rdflib.term.Literal("foo-bar"))
(rdflib.term.URIRef("http://example.org/x"),
 rdflib.term.URIRef("http://example.org/y"),
 rdflib.term.Literal("bar"))
>>>
>>> # querying quads return quads; the fourth argument can be unrestricted
>>> # or restricted to a graph
>>> for q in ds.quads((None, None, None, None)):
...     print(q)
(rdflib.term.URIRef("http://example.org/a"),
 rdflib.term.URIRef("http://www.example.org/b"),
 rdflib.term.Literal("foo"),
 None)
(rdflib.term.URIRef("http://example.org/x"),
 rdflib.term.URIRef("http://example.org/y"),
 rdflib.term.Literal("bar"),
 rdflib.term.URIRef("http://www.example.com/gr"))
(rdflib.term.URIRef("http://example.org/x"),
 rdflib.term.URIRef("http://example.org/z"),
 rdflib.term.Literal("foo-bar"),
 rdflib.term.URIRef("http://www.example.com/gr"))
>>>
>>> for q in ds.quads((None,None,None,g)):
...     print(q)
(rdflib.term.URIRef("http://example.org/x"),
 rdflib.term.URIRef("http://example.org/y"),
```

(continues on next page)

(continued from previous page)

```
 rdflib.term.Literal("bar"),
 rdflib.term.URIRef("http://www.example.com/gr"))
(rdflib.term.URIRef("http://example.org/x"),
 rdflib.term.URIRef("http://example.org/z"),
 rdflib.term.Literal("foo-bar"),
 rdflib.term.URIRef("http://www.example.com/gr"))
>>> # Note that in the call above -
>>> # ds.quads((None,None,None,"http://www.example.com/gr"))
>>> # would have been accepted, too
>>>
>>> # graph names in the dataset can be queried:
>>> for c in ds.graphs():
...     print(c)  # doctest:
DEFAULT
http://www.example.com/gr
>>> # A graph can be created without specifying a name; a skolemized genid
>>> # is created on the fly
>>> h = ds.graph()
>>> for c in ds.graphs():
...     print(c)
DEFAULT
http://rdlib.net/.well-known/genid/rdflib/N...
http://www.example.com/gr
>>> # Note that the Dataset.graphs() call returns names of empty graphs,
>>> # too. This can be restricted:
>>> for c in ds.graphs(empty=False):
...     print(c)
DEFAULT
http://www.example.com/gr
>>>
>>> # a graph can also be removed from a dataset via ds.remove_graph(g)
```

New in version 4.0.

__init__(*store='default'*, *default_union=False*, *default_graph_base=None*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'rdflib.graph'

__str__()
Return str(self).

add_graph(*g*)
alias of graph for consistency

contexts(*triple=None*)
Iterate over all contexts in the graph

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

graph(*identifier=None*, *base=None*)

graphs(*triple=None*)
Iterate over all contexts in the graph

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

parse(*source=None*, *publicID=None*, *format='xml'*, *location=None*, *file=None*, *data=None*, ***args*)
Parse source adding the resulting triples to its own context (sub graph of this graph).

See *rdflib.graph.Graph.parse()* for documentation on arguments.

Returns

The graph into which the source was parsed. In the case of n3 it returns the root context.

quads(*quad*)
Iterate over all the quads in the entire conjunctive graph

remove_graph(*g*)

class rdflib.**Graph**(*store='default'*, *identifier=None*, *namespace_manager=None*, *base=None*)
Bases: *rdflib.term.Node*

An RDF Graph

The constructor accepts one argument, the "store" that will be used to store the graph data (see the "store" package for stores currently shipped with rdflib).

Stores can be context-aware or unaware. Unaware stores take up (some) less space but cannot support features that require context, such as true merging/demerging of sub-graphs and provenance.

The Graph constructor can take an identifier which identifies the Graph by name. If none is given, the graph is assigned a BNode for its identifier.

For more on named graphs, see: http://www.w3.org/2004/03/trix/

__add__(*other*)
Set-theoretic union BNode IDs are not changed.

__and__(*other*)
Set-theoretic intersection. BNode IDs are not changed.

__cmp__(*other*)

__contains__(*triple*)
Support for 'triple in graph' syntax

__dict__ = mappingproxy({'__module__': 'rdflib.graph', '__doc__': 'An RDF Graph\n\n '

__eq__(*other*)
Return self==value.

__ge__(*other*)
Return self>=value.

__getitem__(*item*)
A graph can be "sliced" as a shortcut for the triples method The python slice syntax is (ab)used for specifying triples. A generator over matches is returned, the returned tuples include only the parts not given

```
>>> import rdflib
>>> g = rdflib.Graph()
>>> g.add((rdflib.URIRef("urn:bob"), rdflib.RDFS.label, rdflib.Literal("Bob
↪")))
```

```
>>> list(g[rdflib.URIRef("urn:bob")]) # all triples about bob
[(rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#label'), rdflib.
↪term.Literal('Bob'))]
```

```
>>> list(g[:rdflib.RDFS.label]) # all label triples
[(rdflib.term.URIRef('urn:bob'), rdflib.term.Literal('Bob'))]
```

```
>>> list(g[::rdflib.Literal("Bob")]) # all triples with bob as object
[(rdflib.term.URIRef('urn:bob'), rdflib.term.URIRef('http://www.w3.org/2000/
↪01/rdf-schema#label'))]
```

(continues on next page)

(continued from previous page)

Combined with SPARQL paths, more complex queries can be written concisely:

Name of all Bobs friends:

g[bob : FOAF.knows/FOAF.name]

Some label for Bob:

g[bob : DC.title|FOAF.name|RDFS.label]

All friends and friends of friends of Bob

g[bob : FOAF.knows * "+"]

etc.

New in version 4.0.

__gt__(*other*)
Return self>value.

__hash__()
Return hash(self).

__iadd__(*other*)
Add all triples in Graph other to Graph. BNode IDs are not changed.

__init__(*store='default'*, *identifier=None*, *namespace_manager=None*, *base=None*)
Initialize self. See help(type(self)) for accurate signature.

__isub__(*other*)
Subtract all triples in Graph other from Graph. BNode IDs are not changed.

__iter__()
Iterates over all triples in the store

__le__(*other*)
Return self<=value.

__len__()
Returns the number of triples in the graph

If context is specified then the number of triples in the context is returned instead.

__lt__(*other*)
Return self<value.

__module__ = 'rdflib.graph'

__mul__(*other*)
Set-theoretic intersection. BNode IDs are not changed.

__or__(*other*)
Set-theoretic union BNode IDs are not changed.

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__str__()
Return str(self).

__sub__(*other*)
Set-theoretic difference. BNode IDs are not changed.

__weakref__
list of weak references to the object (if defined)

__xor__(*other*)
Set-theoretic XOR. BNode IDs are not changed.

absolutize(*uri*, *defrag=1*)
Turn uri into an absolute URI if it's not one already

add(*triple*)
Add a triple with self as context

addN(*quads*)
Add a sequence of triple with context

all_nodes()

bind(*prefix*, *namespace*, *override=True*, *replace=False*)
Bind prefix to namespace

If override is True will bind namespace to given prefix even if namespace was already bound to a different prefix.

if replace, replace any existing prefix with the new namespace

for example: graph.bind("foaf", "http://xmlns.com/foaf/0.1/")

close(*commit_pending_transaction=False*)
Close the graph store

Might be necessary for stores that require closing a connection to a database or releasing some resource.

collection(*identifier*)
Create a new `Collection` instance.

Parameters:

- `identifier`: a URIRef or BNode instance.

Example:

```
>>> graph = Graph()
>>> uri = URIRef("http://example.org/resource")
>>> collection = graph.collection(uri)
>>> assert isinstance(collection, Collection)
>>> assert collection.uri is uri
>>> assert collection.graph is graph
>>> collection += [ Literal(1), Literal(2) ]
```

comment(*subject*, *default=''*)
Query for the RDFS.comment of the subject

Return default if no comment exists

commit()
Commits active transactions

compute_qname(*uri*, *generate=True*)

connected()
Check if the Graph is connected

The Graph is considered undirectional.

Performs a search on the Graph, starting from a random node. Then iteratively goes depth-first through the triplets where the node is subject and object. Return True if all nodes have been visited and False if it cannot continue and there are still unvisited nodes left.

de_skolemize(*new_graph=None*, *uriref=None*)

destroy(*configuration*)
Destroy the store identified by *configuration* if supported

property identifier

isomorphic(*other*)
does a very basic check if these graphs are the same If no BNodes are involved, this is accurate.

See rdflib.compare for a correct implementation of isomorphism checks

items(*list*)
Generator over all items in the resource specified by list

list is an RDF collection.

label(*subject*, *default=''*)
Query for the RDFS.label of the subject

Return default if no label exists or any label if multiple exist.

load(*source*, *publicID=None*, *format='xml'*)

n3()
return an n3 identifier for the Graph

property namespace_manager
this graph's namespace-manager

namespaces()
Generator over all the prefix, namespace tuples

objects(*subject=None*, *predicate=None*)
A generator of objects with the given subject and predicate

open(*configuration*, *create=False*)
Open the graph store

Might be necessary for stores that require opening a connection to a database or acquiring some resource.

parse(*source=None*, *publicID=None*, *format=None*, *location=None*, *file=None*, *data=None*, ***args*)
Parse source adding the resulting triples to the Graph.

The source is specified using one of source, location, file or data.

Parameters

- *source*: An InputSource, file-like object, or string. In the case of a string the string is the location of the source.
- *location*: A string indicating the relative or absolute URL of the source. Graph's absolutize method is used if a relative location is specified.
- *file*: A file-like object.
- *data*: A string containing the data to be parsed.
- *format*: Used if format can not be determined from source. Defaults to rdf/xml. Format support can be extended with plugins, but "xml", "n3", "nt" & "trix" are built in.

- *publicID*: the logical URI to use as the document base. If None specified the document location is used (at least in the case where there is a document location).

Returns

- self, the graph instance.

Examples:

```
>>> my_data = '''
... <rdf:RDF
...   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
...   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
... >
...   <rdf:Description>
...     <rdfs:label>Example</rdfs:label>
...     <rdfs:comment>This is really just an example.</rdfs:comment>
...   </rdf:Description>
... </rdf:RDF>
... '''
>>> import tempfile
>>> fd, file_name = tempfile.mkstemp()
>>> f = os.fdopen(fd, "w")
>>> dummy = f.write(my_data)  # Returns num bytes written
>>> f.close()
```

```
>>> g = Graph()
>>> result = g.parse(data=my_data, format="application/rdf+xml")
>>> len(g)
2
```

```
>>> g = Graph()
>>> result = g.parse(location=file_name, format="application/rdf+xml")
>>> len(g)
2
```

```
>>> g = Graph()
>>> with open(file_name, "r") as f:
...     result = g.parse(f, format="application/rdf+xml")
>>> len(g)
2
```

```
>>> os.remove(file_name)
```

predicate_objects(*subject=None*)

A generator of (predicate, object) tuples for the given subject

predicates(*subject=None*, *object=None*)

A generator of predicates with the given subject and object

preferredLabel(*subject*, *lang=None*, *default=None*, *labelProperties=rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'), rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#label')*)

Find the preferred label for subject.

By default prefers skos:prefLabels over rdfs:labels. In case at least one prefLabel is found returns those, else returns labels. In case a language string (e.g., “en”, “de” or even “” for no lang-tagged literals) is given, only such labels will be considered.

Return a list of (labelProp, label) pairs, where labelProp is either skos:prefLabel or rdfs:label.

```
>>> from rdflib import ConjunctiveGraph, URIRef, RDFS, Literal
>>> from rdflib.namespace import SKOS
>>> from pprint import pprint
>>> g = ConjunctiveGraph()
>>> u = URIRef("http://example.com/foo")
>>> g.add([u, RDFS.label, Literal("foo")])
>>> g.add([u, RDFS.label, Literal("bar")])
>>> pprint(sorted(g.preferredLabel(u)))
[(rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#label'),
  rdflib.term.Literal('bar')),
 (rdflib.term.URIRef('http://www.w3.org/2000/01/rdf-schema#label'),
  rdflib.term.Literal('foo'))]
>>> g.add([u, SKOS.prefLabel, Literal("bla")])
>>> pprint(g.preferredLabel(u))
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('bla'))]
>>> g.add([u, SKOS.prefLabel, Literal("blubb", lang="en")])
>>> sorted(g.preferredLabel(u))
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('bla')),
  (rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('blubb', lang='en'))]
>>> g.preferredLabel(u, lang="")
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('bla'))]
>>> pprint(g.preferredLabel(u, lang="en"))
[(rdflib.term.URIRef('http://www.w3.org/2004/02/skos/core#prefLabel'),
  rdflib.term.Literal('blubb', lang='en'))]
```

qname(*uri*)

query(*query_object*, *processor='sparql'*, *result='sparql'*, *initNs=None*, *initBindings=None*, *use_store_provided=True*, ***kwargs*)

Query this graph.

A type of 'prepared queries' can be realised by providing initial variable bindings with initBindings

Initial namespaces are used to resolve prefixes used in the query, if none are given, the namespaces from the graph's namespace manager are used.

Returntype rdflib.query.QueryResult

remove(*triple*)

Remove a triple from the graph

If the triple does not provide a context attribute, removes the triple from all contexts.

resource(*identifier*)

Create a new `Resource` instance.

Parameters:

- `identifier`: a URIRef or BNode instance.

Example:

```
>>> graph = Graph()
>>> uri = URIRef("http://example.org/resource")
>>> resource = graph.resource(uri)
```

(continues on next page)

(continued from previous page)

```
>>> assert isinstance(resource, Resource)
>>> assert resource.identifier is uri
>>> assert resource.graph is graph
```

rollback()
Rollback active transactions

seq(*subject*)
Check if subject is an rdf:Seq

If yes, it returns a Seq class instance, None otherwise.

serialize(*destination=None*, *format='xml'*, *base=None*, *encoding=None*, ***args*)
Serialize the Graph to destination

If destination is None serialize method returns the serialization as a string. Format defaults to xml (AKA rdf/xml).

Format support can be extended with plugins, but "xml", "n3", "turtle", "nt", "pretty-xml", "trix", "trig" and "nquads" are built in.

set(*triple*)
Convenience method to update the value of object

Remove any existing triples for subject and predicate before adding (subject, predicate, object).

skolemize(*new_graph=None*, *bnode=None*, *authority=None*, *basepath=None*)

property store

subject_objects(*predicate=None*)
A generator of (subject, object) tuples for the given predicate

subject_predicates(*object=None*)
A generator of (subject, predicate) tuples for the given object

subjects(*predicate=None*, *object=None*)
A generator of subjects with the given predicate and object

toPython()

transitiveClosure(*func*, *arg*, *seen=None*)
Generates transitive closure of a user-defined function against the graph

```
>>> from rdflib.collection import Collection
>>> g=Graph()
>>> a=BNode("foo")
>>> b=BNode("bar")
>>> c=BNode("baz")
>>> g.add((a,RDF.first,RDF.type))
>>> g.add((a,RDF.rest,b))
>>> g.add((b,RDF.first,RDFS.label))
>>> g.add((b,RDF.rest,c))
>>> g.add((c,RDF.first,RDFS.comment))
>>> g.add((c,RDF.rest,RDF.nil))
>>> def topList(node,g):
...    for s in g.subjects(RDF.rest, node):
...       yield s
>>> def reverseList(node,g):
...    for f in g.objects(node, RDF.first):
...       print(f)
```

(continues on next page)

(continued from previous page)

```
...     for s in g.subjects(RDF.rest, node):
...         yield s
```

```
>>> [rt for rt in g.transitiveClosure(
...     topList,RDF.nil)]
[rdflib.term.BNode('baz'),
 rdflib.term.BNode('bar'),
 rdflib.term.BNode('foo')]
```

```
>>> [rt for rt in g.transitiveClosure(
...     reverseList,RDF.nil)]
http://www.w3.org/2000/01/rdf-schema#comment
http://www.w3.org/2000/01/rdf-schema#label
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
[rdflib.term.BNode('baz'),
 rdflib.term.BNode('bar'),
 rdflib.term.BNode('foo')]
```

transitive_objects(*subject*, *property*, *remember=None*)
Transitively generate objects for the `property` relationship

Generated objects belong to the depth first transitive closure of the `property` relationship starting at `subject`.

transitive_subjects(*predicate*, *object*, *remember=None*)
Transitively generate objects for the `property` relationship

Generated objects belong to the depth first transitive closure of the `property` relationship starting at `subject`.

triples(*triple*)
Generator over the triple store

Returns triples that match the given triple pattern. If triple pattern does not provide a context, all contexts will be searched.

triples_choices(*triple*, *context=None*)

update(*update_object*, *processor='sparql'*, *initNs=None*, *initBindings=None*, *use_store_provided=True*, ***kwargs*)
Update this graph with the given update query.

value(*subject=None*, *predicate=rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#value')*, *object=None*, *default=None*, *any=True*)
Get a value for a pair of two criteria

Exactly one of subject, predicate, object must be None. Useful if one knows that there may only be one value.

It is one of those situations that occur a lot, hence this 'macro' like utility

Parameters: subject, predicate, object – exactly one must be None default – value to be returned if no values found any – if True, return any value in the case there is more than one, else, raise UniquenessError

class rdflib.**ConjunctiveGraph**(*store='default'*, *identifier=None*, *default_graph_base=None*)
Bases: *rdflib.graph.Graph*

A ConjunctiveGraph is an (unnamed) aggregation of all the named graphs in a store.

It has a `default` graph, whose name is associated with the graph throughout its life. *__init__()* can take an identifier to use as the name of this default graph or it will assign a BNode.

All methods that add triples work against this default graph.

All queries are carried out against the union of all graphs.

__contains__(*triple_or_quad*)
Support for 'triple/quad in graph' syntax

__init__(*store='default'*, *identifier=None*, *default_graph_base=None*)
Initialize self. See help(type(self)) for accurate signature.

__len__()
Number of triples in the entire conjunctive graph

__module__ = 'rdflib.graph'

__reduce__()
Helper for pickle.

__str__()
Return str(self).

add(*triple_or_quad*)
Add a triple or quad to the store.

if a triple is given it is added to the default context

addN(*quads*)
Add a sequence of triples with context

context_id(*uri*, *context_id=None*)
URI#context

contexts(*triple=None*)
Iterate over all contexts in the graph

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

get_context(*identifier*, *quoted=False*, *base=None*)
Return a context graph for the given identifier

identifier must be a URIRef or BNode.

parse(*source=None*, *publicID=None*, *format='xml'*, *location=None*, *file=None*, *data=None*, ***args*)
Parse source adding the resulting triples to its own context (sub graph of this graph).

See *rdflib.graph.Graph.parse()* for documentation on arguments.

Returns

The graph into which the source was parsed. In the case of n3 it returns the root context.

quads(*triple_or_quad=None*)
Iterate over all the quads in the entire conjunctive graph

remove(*triple_or_quad*)
Removes a triple or quads

if a triple is given it is removed from all contexts

a quad is removed from the given context only

remove_context(*context*)
Removes the given context from the graph

triples(*triple_or_quad*, *context=None*)
Iterate over all the triples in the entire conjunctive graph

For legacy reasons, this can take the context to query either as a fourth element of the quad, or as the explicit context keyword parameter. The kw param takes precedence.

triples_choices(*triple*, *context=None*)
Iterate over all the triples in the entire conjunctive graph

3.2 Plugins

Many parts of RDFLib are extensible with plugins through setuptools entry-points. These pages list the plugins included in RDFLib core.

3.2.1 Plugin parsers

These serializers are available in default RDFLib, you can use them by passing the name to graph's *parse()* method:

```
graph.parse(my_url, format='n3')
```

The `html` parser will auto-detect RDFa, HTurtle or Microdata.

It is also possible to pass a mime-type for the `format` parameter:

```
graph.parse(my_url, format='application/rdf+xml')
```

If you are not sure what format your file will be, you can use *rdflib.util.guess_format()* which will guess based on the file extension.

Name	Class
html	StructuredDataParser
hturtle	HTurtleParser
mdata	MicrodataParser
microdata	MicrodataParser
n3	*N3Parser*
nquads	*NQuadsParser*
nt	*NTParser*
rdfa	RDFaParser
rdfa1.0	RDFa10Parser
rdfa1.1	RDFaParser
trix	*TriXParser*
turtle	*TurtleParser*
xml	*RDFXMLParser*

3.2.2 Plugin serializers

These serializers are available in default RDFLib, you can use them by passing the name to a graph's *serialize()* method:

```
print graph.serialize(format='n3')
```

It is also possible to pass a mime-type for the `format` parameter:

```
graph.serialize(my_url, format='application/rdf+xml')
```

Name	Class
n3	*N3Serializer*
nquads	*NQuadsSerializer*
nt	*NTSerializer*
pretty-xml	*PrettyXMLSerializer*
trig	*TrigSerializer*
trix	*TriXSerializer*
turtle	*TurtleSerializer*
xml	*XMLSerializer*

3.2.3 Plugin stores

Name	Class
Auditable	*AuditableStore*
Concurrent	*ConcurrentStore*
IOMemory	*IOMemory*
SPARQLStore	*SPARQLStore*
SPARQLUpdateStore	*SPARQLUpdateStore*
Sleepycat	*Sleepycat*
default	*IOMemory*

3.2.4 Plugin query results

Plugins for reading and writing of (SPARQL) `QueryResult` - pass `name` to either `parse()` or `serialize()`

Parsers

Name	Class
csv	*CSVResultParser*
json	*JSONResultParser*
tsv	*TSVResultParser*
xml	*XMLResultParser*

Serializers

Name	Class
csv	*CSVResultSerializer*
json	*JSONResultSerializer*
txt	*TXTResultSerializer*
xml	*XMLResultSerializer*

CHAPTER

FOUR

FOR DEVELOPERS

4.1 RDFLib developers guide

4.1.1 Introduction

This document describes the process and conventions to follow when developing RDFLib code.

Please be as Pythonic as possible (**PEP 8**).

Code will occasionally be auto-formatted using `autopep8` - you can also do this yourself.

Any new functionality being added to RDFLib should have doc tests and unit tests. Tests should be added for any functionality being changed that currently does not have any doc tests or unit tests. And all the tests should be run before committing changes to make sure the changes did not break anything.

If you add a new cool feature, consider also adding an example in `./examples`

4.1.2 Running tests

Run tests with nose:

Specific tests can either be run by module name or file name. For example:

```
$ python run_tests.py --tests rdflib.graph
$ python run_tests.py --tests test/test_graph.py
```

4.1.3 Writing documentation

We use sphinx for generating HTML docs, see *Writing RDFLib Documentation*

4.1.4 Continous Integration

We used Travis for CI, see:

https://travis-ci.org/RDFLib/rdflib

If you make a pull-request to RDFLib on GitHub, travis will automatically test you code.

4.1.5 Compatibility

RDFLib>=5.0.0 tries to be compatible with python versions 2.7, 3.5, 3.6, 3.7.

4.1.6 Releasing

Set to-be-released version number in `rdflib/__init__.py` and `README.md`. Check date in `LICENSE`.

Add `CHANGELOG.md` entry.

Commit this change. It's preferable make the release tag via https://github.com/RDFLib/rdflib/releases/new :: Our Tag versions aren't started with 'v', so just use a plain 5.0.0 like version. Release title is like "RDFLib 5.0.0", the description a copy of your `CHANGELOG.md` entry. This gives us a nice release page like this:: https://github.com/RDFLib/rdflib/releases/tag/4.2.2

If for whatever reason you don't want to take this approach, the old one is:

```
Tagging the release commit with::

  git tag -a -m 'tagged version' X.X.X

When pushing, remember to do::

  git push --tags
```

No matter how you create the release tag, remember to upload tarball to pypi with:

```
rm -r dist/X.X.X[.-]*  # delete all previous builds for this release, just in case

rm -r build
python setup.py sdist
python setup.py bdist_wheel
ls dist

# upload with twine
# WARNING: once uploaded can never be modified, only deleted!
twine upload dist/rdflib-X.X.X[.-]*
```

Set new dev version number in the above locations, i.e. next release *-dev*: `5.0.1-dev` and commit again.

Tweet, email mailing list and update the topic of #rdflib on freenode irc:

```
/msg ChanServ topic #rdflib https://github.com/RDFLib/rdflib | latest stable version:␣
↪4.2.0 | docs: http://rdflib.readthedocs.org
```

4.2 Writing RDFLib Documentation

The docs are generated with Sphinx.

Sphinx makes it very easy to pull in doc-strings from modules, classes, methods, etc. When writing doc-strings, special reST fields can be used to annotate parameters, return-types, etc. This make for pretty API docs:

http://sphinx-doc.org/domains.html?highlight=param#info-field-lists

4.2.1 Building

To build you must have the *sphinx* package installed:

```
pip install sphinx
```

Then you can do:

```
python setup.py build_sphinx
```

The docs will be generated in `build/sphinx/html/`

4.2.2 API Docs

API Docs are automatically generated with `sphinx-apidoc`:

```
sphinx-apidoc -f -d 10 -o docs/apidocs/ rdflib examples
```

(then `rdflib.rst` was tweaked manually to not include all convenience imports that are directly in the `rdflib/__init__.py`)

4.2.3 Tables

The tables in `plugin_*.rst` were generated with `plugintable.py`

4.3 A Universal RDF Store Interface

This document attempts to summarize some fundamental components of an RDF store. The motivation is to outline a standard set of interfaces for providing the support needed to persist an RDF Graph in a way that is universal and not tied to any specific implementation.

For the most part, the interfaces adhere to the core RDF model and use terminology that is consistent with the RDF Model specifications. However, these suggested interfaces also extends an RDF store with additional requirements necessary to facilitate those aspects of Notation 3 that go beyond the RDF model to provide a framework for First Order Predicate Logic processing and persistence.

4.3.1 Terminology

Context

A named, unordered set of statements (that could also be called a sub-graph). The `named graph` literature and ontology are relevant to this concept. The term `context` could be thought of as either the sub-graph itself or the relationship between an RDF triple and a sub-graph in which it is found (this latter is how the term context is used in the Notation 3 Design Issues page).

It is worth noting that the concept of logically grouping triples within an addressable 'set' or 'subgraph' is just barely beyond the scope of the RDF model. The RDF model defines a graph to be an arbitrary collection of triples and the semantics of these triples — but doesn't give guidance on how to address such arbitrary collections in a consistent manner. Although a collection of triples can be thought of as a resource itself, the association between a

triple and the collection (of which it is a part) is not covered. Public RDF is an example of an attempt to formally model this relationship - and includes one other unrelated extension: Articulated Text

Conjunctive Graph

This refers to the 'top-level' Graph. It is the aggregation of all the contexts within it and is also the appropriate, absolute boundary for closed world assumptions / models. This distinction is the low-hanging fruit of RDF along the path to the semantic web and most of its value is in (corporate/enterprise) real-world problems:

> There are at least two situations where the closed world assumption is used. The first is where it is assumed that a knowledge base contains all relevant facts. This is common in corporate databases. That is, the information it contains is assumed to be complete

From a store perspective, closed world assumptions also provide the benefit of better query response times, due to the explicit closed world boundaries. Closed world boundaries can be made transparent by federated queries that assume each `ConjunctiveGraph` is a section of a larger, unbounded universe. So a closed world assumption does not preclude you from an open world assumption.

For the sake of persistence, Conjunctive Graphs must be distinguished by identifiers (which may not necessarily be RDF identifiers or may be an RDF identifier normalized - SHA1/MD5 perhaps - for database naming purposes) that could be referenced to indicate conjunctive queries (queries made across the entire conjunctive graph) or appear as nodes in asserted statements. In this latter case, such statements could be interpreted as being made about the entire 'known' universe. For example:

```
<urn:uuid:conjunctive-graph-foo> rdf:type :ConjunctiveGraph
<urn:uuid:conjunctive-graph-foo> rdf:type log:Truth
<urn:uuid:conjunctive-graph-foo> :persistedBy :MySQL
```

Quoted Statement

A statement that isn't asserted but is referred to in some manner. Most often, this happens when we want to make a statement about another statement (or set of statements) without necessarily saying these quoted statements (are true). For example:

```
Chimezie said "higher-order statements are complicated"
```

Which can be written (in N3) as:

```
:chimezie :said {:higherOrderStatements rdf:type :complicated}
```

Formula

A context whose statements are quoted or hypothetical.

Context quoting can be thought of as very similar to reification. The main difference is that quoted statements are not asserted or considered as statements of truth about the universe and can be referenced as a group: a hypothetical RDF Graph

Universal Quantifiers / Variables

(relevant references):

- OWL Definition of SWRL.
- SWRL/RuleML Variable

Terms

Terms are the kinds of objects that can appear in a quoted/asserted triple.

This includes those that are core to RDF:

- Blank Nodes
- URI References
- Literals (which consist of a literal value, datatype and language tag)

Those that extend the RDF model into N3:

- Formulae
- Universal Quantifications (Variables)

And those that are primarily for matching against 'Nodes' in the underlying Graph:

- REGEX Expressions
- Date Ranges
- Numerical Ranges

Nodes

Nodes are a subset of the Terms that the underlying store actually persists. The set of such Terms depends on whether or not the store is formula-aware. Stores that aren't formula-aware would only persist those terms core to the RDF Model, and those that are formula-aware would be able to persist the N3 extensions as well. However, utility terms that only serve the purpose for matching nodes by term-patterns probably will only be terms and not nodes.

The set of nodes of an RDF graph is the set of subjects and objects of triples in the graph.

Context-aware

An RDF store capable of storing statements within contexts is considered context-aware. Essentially, such a store is able to partition the RDF model it represents into individual, named, and addressable sub-graphs.

Formula-aware

An RDF store capable of distinguishing between statements that are asserted and statements that are quoted is considered formula-aware.

Such a store is responsible for maintaining this separation and ensuring that queries against the entire model (the aggregation of all the contexts - specified by not limiting a 'query' to a specifically name context) do not include quoted statements. Also, it is responsible for distinguishing universal quantifiers (variables).

Note: These 2 additional concepts (formulae and variables) must be thought of as core extensions and distinguishable from the other terms of a triple (for the sake of the persistence round trip - at the very least). It's worth noting that the 'scope' of universal quantifiers (variables) and existential quantifiers (BNodes) is the formula (or context - to be specific) in which their statements reside. Beyond this, a Formula-aware store behaves the same as a Context-aware store.

Conjunctive Query

Any query that doesn't limit the store to search within a named context only. Such a query expects a context-aware store to search the entire asserted universe (the conjunctive graph). A formula-aware store is expected not to include quoted statements when matching such a query.

N3 Round Trip

This refers to the requirements on a formula-aware RDF store's persistence mechanism necessary for it to be properly populated by a N3 parser and rendered as syntax by a N3 serializer.

Transactional Store

An RDF store capable of providing transactional integrity to the RDF operations performed on it.

4.3.2 Interpreting Syntax

The following Notation 3 document:

```
{ ?x a :N3Programmer } => { ?x :has [a :Migraine] }
```

Could cause the following statements to be asserted in the store:

```
_:a log:implies _:b
```

This statement would be asserted in the partition associated with quoted statements (in a formula named `_:a`)

```
?x rdf:type :N3Programmer
```

Finally, these statements would be asserted in the same partition (in a formula named `_:b`)

```
?x :has _:c

_:c rdf:type :Migraine
```

4.3.3 Formulae and Variables as Terms

Formulae and variables are distinguishable from URI references, Literals, and BNodes by the following syntax:

```
{ .. } - Formula ?x - Variable
```

They must also be distinguishable in persistence to ensure they can be round-tripped.

Note: There are a number of other issues regarding the *Persisting Notation 3 Terms*.

4.3.4 Database Management

An RDF store should provide standard interfaces for the management of database connections. Such interfaces are standard to most database management systems (Oracle, MySQL, Berkeley DB, Postgres, etc..)

The following methods are defined to provide this capability (see below for description of the *configuration* string):

`Store.`**`open`**(*configuration*, *create=False*)
: Opens the store specified by the configuration string. If create is True a store will be created if it does not already exist. If create is False and a store does not already exist an exception is raised. An exception is also raised if a store exists, but there is insufficient permissions to open the store. This should return one of: VALID_STORE, CORRUPTED_STORE, or NO_STORE

`Store.`**`close`**(*commit_pending_transaction=False*)
: This closes the database connection. The commit_pending_transaction parameter specifies whether to commit all pending transactions before closing (if the store is transactional).

`Store.`**`destroy`**(*configuration*)
: This destroys the instance of the store identified by the configuration string.

The *configuration* string is understood by the store implementation and represents all the parameters needed to locate an individual instance of a store. This could be similar to an ODBC string or in fact be an ODBC string, if the connection protocol to the underlying database is ODBC.

The *open()* function needs to fail intelligently in order to clearly express that a store (identified by the given configuration string) already exists or that there is no store (at the location specified by the configuration string) depending on the value of `create`.

4.3.5 Triple Interfaces

An RDF store could provide a standard set of interfaces for the manipulation, management, and/or retrieval of its contained triples (asserted or quoted):

`Store.`**`add`**(*triple*, *context*, *quoted=False*)
: Adds the given statement to a specific context or to the model. The quoted argument is interpreted by formula-aware stores to indicate this statement is quoted/hypothetical It should be an error to not specify a context and have the quoted argument be True. It should also be an error for the quoted argument to be True when the store is not formula-aware.

`Store.`**`remove`**(*triple*, *context=None*)
: Remove the set of triples matching the pattern from the store

`Store.`**`triples`**(*triple_pattern*, *context=None*)
: A generator over all the triples matching the pattern. Pattern can include any objects for used for comparing against nodes in the store, for example, REGEXTerm, URIRef, Literal, BNode, Variable, Graph, QuotedGraph, Date? DateRange?

Parameters **context** – A conjunctive query can be indicated by either providing a value of None, or a specific context can be queries by passing a Graph instance (if store is context aware).

Note: The *triples()* method can be thought of as the primary mechanism for producing triples with nodes that match the corresponding terms in the `(s, p, o)` term pattern provided. The term pattern `(None, None, None)` matches *all* nodes.

`Store.__len__`(*context=None*)
Number of statements in the store. This should only account for non- quoted (asserted) statements if the context is not specified, otherwise it should return the number of statements in the formula or context given.

Parameters **context** – a graph instance to query or None

4.3.6 Formula / Context Interfaces

These interfaces work on contexts and formulae (for stores that are formula-aware) interchangeably.

`ConjunctiveGraph.contexts`(*triple=None*)
Iterate over all contexts in the graph

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

`ConjunctiveGraph.remove_context`(*context*)
Removes the given context from the graph

4.3.7 Interface Test Cases

Basic

Tests parsing, triple patterns, triple pattern removes, size, contextual removes

Source Graph

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://test/> .
{:a :b :c; a :foo} => {:a :d :c} .
_:foo a rdfs:Class .
:a :d :c .
```

Test code

```
implies = URIRef("http://www.w3.org/2000/10/swap/log#implies")
a = URIRef('http://test/a')
b = URIRef('http://test/b')
c = URIRef('http://test/c')
d = URIRef('http://test/d')
for s,p,o in g.triples((None,implies,None)):
    formulaA = s
    formulaB = o
```

(continues on next page)

(continued from previous page)

```
    #contexts test
    assert len(list(g.contexts()))==3

    #contexts (with triple) test
    assert len(list(g.contexts((a,d,c))))==2

    #triples test cases
    assert type(list(g.triples((None,RDF.type,RDFS.Class)))[0][0]) == BNode
    assert len(list(g.triples((None,implies,None))))==1
    assert len(list(g.triples((None,RDF.type,None))))==3
    assert len(list(g.triples((None,RDF.type,None),formulaA)))==1
    assert len(list(g.triples((None,None,None),formulaA)))==2
    assert len(list(g.triples((None,None,None),formulaB)))==1
    assert len(list(g.triples((None,None,None))))==5
    assert len(list(g.triples((None,URIRef('http://test/d'),None),formulaB)))==1
    assert len(list(g.triples((None,URIRef('http://test/d'),None))))==1

    #Remove test cases
    g.remove((None,implies,None))
    assert len(list(g.triples((None,implies,None))))==0
    assert len(list(g.triples((None,None,None),formulaA)))==2
    assert len(list(g.triples((None,None,None),formulaB)))==1
    g.remove((None,b,None),formulaA)
    assert len(list(g.triples((None,None,None),formulaA)))==1
    g.remove((None,RDF.type,None),formulaA)
    assert len(list(g.triples((None,None,None),formulaA)))==0
    g.remove((None,RDF.type,RDFS.Class))

    #remove_context tests
    formulaBContext=Context(g,formulaB)
    g.remove_context(formulaB)
    assert len(list(g.triples((None,RDF.type,None))))==2
    assert len(g)==3 assert len(formulaBContext)==0
    g.remove((None,None,None))
    assert len(g)==0
```

Formula and Variables Test

Source Graph

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://test/> .
{?x a rdfs:Class} => {?x a :Klass} .
```

Test Code

```
implies = URIRef("http://www.w3.org/2000/10/swap/log#implies")
klass = URIRef('http://test/Klass')
for s,p,o in g.triples((None,implies,None)):
    formulaA = s
    formulaB = o
    assert type(formulaA) == Formula
    assert type(formulaB) == Formula
    for s,p,o in g.triples((None,RDF.type,RDFS.Class)),formulaA):
        assert type(s) == Variable
    for s,p,o in g.triples((None,RDF.type,klass)),formulaB):
        assert type(s) == Variable
```

Transactional Tests

To be instantiated.

4.3.8 Additional Terms to Model

These are a list of additional kinds of RDF terms (all of which are special Literals)

- `rdflib.plugins.store.regexmatching.REGEXTerm` - a REGEX string which can be used in any term slot in order to match by applying the Regular Expression to statements in the underlying graph.
- Date (could provide some utility functions for date manipulation / serialization, etc..)
- DateRange

4.3.9 Namespace Management Interfaces

The following namespace management interfaces (defined in Graph) could be implemented in the RDF store. Currently, they exist as stub methods of *`Store`* and are defined in the store subclasses (e.g. `IOMemory`):

`Store.`**`bind`**`(`*prefix*, *namespace*`)`

`Store.`**`prefix`**`(`*namespace*`)`

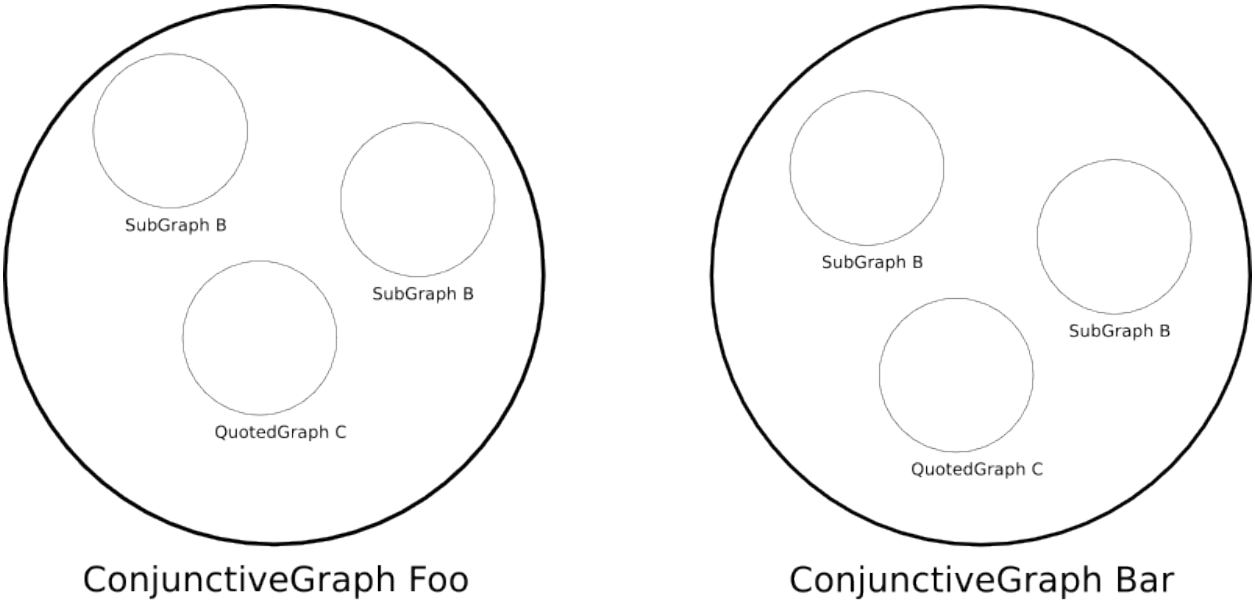
`Store.`**`namespace`**`(`*prefix*`)`

`Store.`**`namespaces`**`()`

4.3.10 Open issues

Does the Store interface need to have an identifier property or can we keep that at the Graph level?

The Store implementation needs a mechanism to distinguish between triples (quoted or asserted) in ConjunctiveGraphs (which are mutually exclusive universes in systems that make closed world assumptions - and queried separately). This is the separation that the store identifier provides. This is different from the name of a context within a ConjunctiveGraph (or the default context of a conjunctive graph). I tried to diagram the logical separation of ConjunctiveGraphs, SubGraphs and QuotedGraphs in this diagram



An identifier of `None` can be used to indicate the store (aka *all contexts*) in methods such as *`triples()`*, *`__len__()`*, etc. This works as long as we're only dealing with one Conjunctive Graph at a time – which may not always be the case.

Is there any value in persisting terms that lie outside N3 (`rdflib.plugins.store.regexmatching.REGEXTerm`, Date, etc..)?

Potentially, not sure yet.

Should a conjunctive query always return quads instead of triples? It would seem so, since knowing the context that produced a triple match is an essential aspect of query construction / optimization. Or if having the triples function yield/produce different length tuples is problematic, could an additional - and slightly redundant - interface be introduced?:

`ConjunctiveGraph.`**`quads`**(*triple_or_quad=None*)
Iterate over all the quads in the entire conjunctive graph

Stores that weren't context-aware could simply return `None` as the 4th item in the produced/yielded tuples or simply not support this interface.

4.4 Persisting Notation 3 Terms

4.4.1 Using N3 Syntax for Persistence

Blank Nodes, Literals, URI References, and Variables can be distinguished in persistence by relying on Notation 3 syntax convention.

All URI References can be expanded and persisted as:

```
<..URI..>
```

All Literals can be expanded and persisted as:

```
"..value.."@lang or "..value.."^^dtype_uri
```

Note: `@lang` is a language tag and `^^dtype_uri` is the URI of a data type associated with the Literal

Blank Nodes can be expanded and persisted as:

```
_:Id
```

Note: where Id is an identifier as determined by skolemization. Skolemization is a syntactic transformation routinely used in automatic inference systems in which existential variables are replaced by 'new' functions - function names not used elsewhere - applied to any enclosing universal variables. In RDF, Skolemization amounts to replacing every blank node in a graph by a 'new' name, i.e. a URI reference which is guaranteed to not occur anywhere else. In effect, it gives 'arbitrary' names to the anonymous entities whose existence was asserted by the use of blank nodes: the arbitrariness of the names ensures that nothing can be inferred that would not follow from the bare assertion of existence represented by the blank node. (Using a literal would not do. Literals are never 'new' in the required sense.)

Variables can be persisted as they appear in their serialization `(?varName)` - since they only need be unique within their scope (the context of their associated statements)

These syntactic conventions can facilitate term round-tripping.

4.4.2 Variables by Scope

Would an interface be needed in order to facilitate a quick way to aggregate all the variables in a scope (given by a formula identifier)? An interface such as:

```
def variables(formula_identifier)
```

4.4.3 The Need to Skolemize Formula Identifiers

It would seem reasonable to assume that a formula-aware store would assign Blank Node identifiers as names of formulae that appear in a N3 serialization. So for instance, the following bit of N3:

```
{?x a :N3Programmer} => {?x :has :Migrane}
```

Could be interpreted as the assertion of the following statement:

```
_:a log:implies _:b
```

However, how are `_:a` and `_:b` distinguished from other Blank Nodes? A formula-aware store would be expected to persist the first set of statements as quoted statements in a formula named `_:a` and the second set as quoted statements in a formula named `_:b`, but it would not be cost-effective for a serializer to have to query the store for all statements in a context named `_:a` in order to determine if `_:a` was associated with a formula (so that it could be serialized properly).

4.4.4 Relying on `log:Formula` Membership

The store could rely on explicit `log:Formula` membership (via `rdf:type` statements) to model the distinction of Blank Nodes associated with formulae. However, would these statements be expected from an N3 parser or known implicitly by the store? i.e., would all such Blank Nodes match the following pattern:

```
?formula rdf:type log:Formula
```

4.4.5 Relying on an Explicit Interface

A formula-aware store could also support the persistence of this distinction by implementing a method that returns an iterator over all the formulae in the store:

```
def formulae(triple=None)
```

This function would return all the Blank Node identifiers assigned to formulae or just those that contain statements matching the given triple pattern and would be the way a serializer determines if a term refers to a formula (in order to properly serializer it).

How much would such an interface reduce the need to model formulae terms as first class objects (perhaps to be returned by the `triple()` function)? Would it be more useful for the *Graph* (or the store itself) to return a Context object in place of a formula term (using the formulae interface to make this determination)?

Conversely, would these interfaces (variables and formulae) be considered optimizations only since you have the distinction by the kinds of terms triples returns (which would be expanded to include variables and formulae)?

4.4.6 Persisting Formula Identifiers

This is the most straight forward way to maintain this distinction - without relying on extra interfaces. Formula identifiers could be persisted distinctly from other terms by using the following notation:

```
{_:bnode} or {<.. URI ..>}
```

This would facilitate their persistence round-trip - same as the other terms that rely on N3 syntax to distinguish between each other.

Developers might also like to join rdflib's dev mailing list: https://groups.google.com/group/rdflib-dev

CHAPTER

FIVE

THE CODE

The rdflib code is hosted on GitHub at https://github.com/RDFLib/rdflib where you lodge Issues and also Pull Requests to help improve this community project!

The RDFlib organisation on GitHub at https://github.com/RDFLib maintains this package and a number of other RDF and related packaged that you might also find useful.

CHAPTER

SIX

FURTHER HELP

For asynchronous chat support, try our gitter channel at https://gitter.im/RDFLib/rdflib

If you would like more help with using rdflib, please post a question using the tag `[rdflib]` on StackOverflow. A list of existing `[rdflib]` tagged questions is there at:

- https://stackoverflow.com/questions/tagged/rdflib

PYTHON MODULE INDEX

e

r

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X