Lab Session #02

Introduction

Welcome to the second lab. This time, we'll start programming knowledge graphs using Python.

Task #1: RDF

Your first task is to translate some of the knowledge graphs you developed on last week's Worksheet #1 into a real RDF graph. Write down the triples using the Turtle format discussed in the lecture. Then, validate your graph by:

1. Using a browser, go to http://ttl.summerofcode.be and paste your Turtle code into the designated text area.
2. Click the “Validate! ” button.
3. Examine the results of parsing the input. Correct any mistakes that you might have made accordingly.
4. If no mistakes are found in the input, you should see a message that reads “Congrats! Your syntax is correct.”.

There are a number of other RDF-related tools online; for example, try out the RDF converter at http://rdfvalidator.mybluemix.net/ and convert your Turtle file (.ttl) into JSON-LD and RDF/XML to get an idea how these formats look like. The validator at https://www.w3.org/RDF/Validator/ only accepts RDF/XML, but it can additionally draw you a graph corresponding to your triples (under the "Display Result Options", select "Triples and Graph"). Convert your RDF file from Turtle to RDF/XML and visualize it in form of a graph.

Note: you will probably encounter references to `rdfs:` (RDF Schema) in examples you find online; we will cover the details of RDFS in this week's lecture.

Task #2: RDFlib

For working with RDF and related standards, there are a multitude of libraries available. For example, a popular open source framework for *Java* is Apache Jena. Here, we will use the RDFlib for Python (documentation).

First, we need to install RDFLib. Run the command below in your Conda environment:

```
pip install rdflib
```

and check out the first section in the documentation, Getting started with RDFLib.

Task #2.1 First steps with RDFlib

Now, we want to load the graph you prepared in Task 1. The code you need from RDFLib is `<graph>.parse`, as shown below:

```
import rdflib

# Create a Graph
g = rdflib.Graph()

# Parse an RDF file
g.parse(<RDFFile_Path>) # Provide the RDF file path here
```

You can now print out your whole graph `g` using the code below:

```
# Loop through each triple in the graph (subj, pred, obj)
for s,p,o in g:
    # Print the subject, predicate and the object
    print s,p,o
```

Now, go through the RDFLib documentation section, Loading and saving RDF to see how to read and write RDF graphs in different formats. Add code to write the triples in a different format, e.g., RDF/XML and N-Triples.

Task #2.2 Creating triples and namespaces

We can also create triples directly with RDFLib. To begin, read the documentation section, Creating RDF triples.

Now, start your program by importing the following libraries:

```
from rdflib import Graph, Literal, RDF, Namespace
from rdflib.namespace import FOAF, RDFS
```

Then, create a new graph instance with:

```
g = Graph()
```

Now you can use the `g.add()` function to add triples to the graph. Write code that adds triples representing:

- `<Joe> <is a> <foaf:Person>`
- `<Joe> <rdfs:label> "Joe"`
- `<Joe> <foaf:knows> <Jane>`

For printing the generated graph, you can use the `g.serialize()` function.

Note: RDFLib has some pre-defined namespaces that can be used for graph creation. In order to add another prefix to the knowledge graph, like `foaf`, use the `g.bind()` function. See Namespaces and Bindings for the details.

Task #2.3 Navigating Graphs

The next step is to learn how to *navigate* graphs, in order to retrieve the knowledge we need, for example, to answer a question.

RDFLib supports basic *triple pattern matching* through the `triples()` function. You can match specific parts of a graph with the methods `objects(), subjects(), predicates()`, etc. Look at the documentation section on *"Navigating Graphs"* for the details.

Now, write a Python program that prints out all triples in your graph for the subject URI corresponding to `Joe`.

Note: The SPARQL graph query language will be covered in Lecture #4 and its corresponding lab.

Task #2.4 Merging Graphs

A powerful feature of knowledge graphs is that we can *merge* different graphs together, in order to connect and integrate knowledge from different sources. In this task, we will merge the graph you created above with the knowledge about Concordia from DBpedia.

RDFLib supports various graph operations, like union (addition), intersection, difference (subtraction) and XOR. Like before, we first import RDFLib and create a graph instance, which we'll call <code>g1</code>:

```
import rdflib
# Create graph g1
g1 = rdflib.Graph()
```

We can now parse the DBpedia graph, by directly loading it from the online knowledge base URI:

```
# Parse the DBpedia graph about Concordia
g1.parse("https://dbpedia.org/resource/Concordia_University")
```

To check the number of triples in our DBpedia graph about Concordia University, use the command below. The output should show a count of about ~2300:

```
# Print the number of triples in the graph
print(len(g1))
```

Similarly, load the graph your create above in Task 1 using the `g.parse()` function. To merge the two graphs (union), use the addition (+) operator.

Print out the merged graph using the `g.serialize()` function and verify that they were indeed merged using Concordia's URI.

Note that you can now answer questions from this merged graph that were not possible to answer from each of them alone, for example, *"In which city is the university located that Joe is studying at?"'*

That's all for this lab!