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 (use `.ttl` for the file extension, like in `mytriples.ttl`). 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 https://issemantic.net/rdf-converter 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.

Some notes to get started:

- **Turtle:** If you're unsure about the Turtle format, best look at the examples on the lecture slides and the RDF Primer (the full details are defined here, but that's not needed to get started).
- **RDFS:** 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 #3.
- **Predicates:** We have not yet covered the details how predicates like "studies at" are encoded in URIs. The details will come in the next lecture & lab; for now, just use a URI like `http://example.org/studiesAt`.
- **Wikidata:** Remember to use the correct form for Wikidata URIs using the `entity/` path, e.g., http://www.wikidata.org/entity/Q326342 for Concordia. The `wiki/` path is a redirect for displaying a human-readable HTML page in a browser and cannot be used for working with RDF triples in a program.
- **Namespace Declarations:** Remember to declare necessary namespaces at the beginning of your Turtle files. This is often overlooked by beginners and can lead to validation errors.
- **Examples of Common Mistakes:** Watch out for common pitfalls such as incorrect URI formats, forgetting to close triples with a period, or mismatched prefixes. We'll review some of these in class, but keeping an eye out for them now can save you time.
- **Encouragement for Peer Review:** Consider reviewing a classmate's Turtle file and having them review yours. Peer review can be an excellent way to learn from each other and catch errors you might have missed.
- **Reminder on File Extension:** Make sure to save your work with the `.ttl` file extension. This ensures that the tools we use can properly recognize and process your Turtle files.

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("path/to/your/file.ttl") # 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.

Note: If you encounter any issues with installation or loading files, refer to the troubleshooting section in the RDFLib documentation, or ask for help in class.

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>`

Note: Remember to replace `<Joe>` and `<Jane>` with appropriate URIs to represent these individuals.

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

Additional notes:

- RDFLib includes pre-defined namespaces like FOAF and RDFS, which are commonly used in RDF graphs. FOAF, for instance, is used to describe people and their relationships.
- In order to add another prefix to the knowledge graph, like `foaf`, use the `g.bind()` function. This not only adds the prefix but also makes your Turtle file more readable by avoiding full URIs. 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.

Your task: Write a Python program that prints out all triples in your graph for the subject URI corresponding to `Joe`.

Notes:

- Experiment with `objects()`, `subjects()`, and `predicates()` methods to see different ways of accessing graph data.
- Ensure the URI for Joe is consistent with how you've represented it in Task 2.2.
- The SPARQL graph query language, which provides more sophisticated ways of querying graphs, 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 `g1`:

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

Now, let's load information about Concordia University from Wikidata into our graph. We can parse the Wikidata graph by directly loading it from the following online knowledge base URI:

```
# Parse the Wikidata graph about Concordia (in Turtle format)
g1.parse("http://www.wikidata.org/entity/Q326342.ttl")
```

To check the number of triples in our DBpedia graph about Concordia University, use the command below. The output should show a large number of triples, more than ~3500:

```
# 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?"*. Try running your code from Question 2.1 on this merged graph to see all the information you now have available. Then, write some triple patterns to print out the city where Joe is studying.

You can also experiment with merging the data about Concordia from the DBpedia knowledge graph using:

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

This exercise serves as an introduction to the expansive capabilities of knowledge graphs in AI systems. By further exploring linked data, such as loading additional details about Montreal (Q340) from Wikidata, an autonomous agent can enhance its understanding of related concepts like the country in which the city is located.

Closing notes: Industrial Relevance of Knowledge Graphs

1. **Enhanced Data Integration:** In the business world, where companies often grapple with vast, diverse data sets, knowledge graphs play a crucial role in integrating this data cohesively. This integration allows for more effective data management, crucial for making informed decisions and streamlining operations across various departments.
2. **Advanced Analytics and Insights:** Knowledge graphs enable companies to delve deeper into analytics, offering richer insights into customer behavior, market trends, and operational efficiencies. These insights are invaluable for strategic decision-making, helping businesses stay ahead in competitive markets.
3. **AI and Machine Learning Enhancement:** For companies investing in AI and machine learning, knowledge graphs provide essential context and relational data, significantly improving the accuracy of predictive models and algorithms. This enhancement is particularly vital in areas like natural language processing and intelligent assistants, key drivers of innovation and efficiency in many industries.

Incorporating knowledge graphs allows businesses to manage and leverage their data more effectively, leading to smarter strategies, enhanced customer experiences, and a competitive edge in today's data-driven economy.

That's all for this lab!