This lab sheet and all other materials can be found on GitHub here: https://github.com/QMUL-ECS735P/lab-week-4/

Introduction to SPARQL

> **Session objectives:**
>
> - Write and execute SPARQL queries
> - Merge RDF data from multiple sources

1. Introduction

As with the last lab, we'll be using Python to create SPARQL database queries and constructing RDF graphs from those queries.

If you're new to Python it would be wise to keep the basic tutorial handy for reference:

- https://docs.python.org/3.7/tutorial/.

Python scripts are run by calling the `python` command, followed by the name of the python script to execute, for example:

```
python3 hello_world.py
```

It is also possible to run python in an interactive mode known as a REPL, to do this simply call `python` without any arguments:

```
python3
```

If you are using one of the machines in the ITL, you will be able to follow along with this lab sheet with minimal fuss. Make sure you boot into **Linux** and not Windows. If you would like to use your own computer, you'll need to make sure you have python installed as well as the following additional libraries:

- sparql-wrapper: https://github.com/RDFLib/sparqlwrapper
- rdflib: https://github.com/RDFLib/rdflib
- pyparsing: http://pyparsing.wikispaces.com/
- networkx: http://networkx.lanl.gov/

To run the interactive notebooks, you will also need to install Jupyter: https://jupyter.org/

Along with this lab sheet you should also find `using-itl.txt` and `using-laptop.txt` which will run through the steps necessary to get things working on either the ITL computers or your own laptop.

2. Writing SPARQL Queries

In this lab we'll be using an eternal API endpoint hosted by DBpedia. DBpedia is a community-run resource that extracts linked data from Wikipedia.

To write SPARQL queries as python strings, we'll be making good use of multi-line strings:

```
this is
a multiline
string
```

Begin by importing the SPARQLWrapper library, and supressing some warnings to make the output clearer:

```
from SPARQLWrapper import SPARQLWrapper, JSON
import warnings

warnings.filterwarnings('ignore')
```

DBpedia has a SPARQL endpoint located at `http://dbpedia.org/sparql` . We're going to wrap that endpoint using the SPARQLWrapper, which means we can use a lot of handy methods to construct, exectue, and receive queries.

```
endpoint = SPARQLWrapper('http://dbpedia.org/sparql')
```

Now we're going to write a simple SPARQL query to get some information from the database:

```
endpoint.setQuery('''
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpr: <http://dbpedia.org/resource/>
SELECT ?label
WHERE { dbpr:Asturias rdfs:label ?label }
```

```
''')
```

Before we execute the query, look at how the query is constructed. Does it look familiar?

The endpoint can respond in a variety of different serialisation formats, for today we'll be using JSON.

```
endpoint.setReturnFormat(JSON)
```

Now to execute the query and print the results:

```
results = endpoint.query().convert()

for result in results['results']['bindings']:
  print(result['label']['value'])
```

Let's take a moment to unpack what's happening here. First we execute the query with `endpoint.query()` and then tell the SPARQLWrapper library to convert the response into Python nested dictionaries with `.convert()`.

We then iterate over the array contained in `results['results']['bindings']` and print each value contained in `result['label']['value']`.

To get a better idea of how the `results` dictionary is structured we can use the `pprint` library to pretty print the data.

```
from pprint import PrettyPrinter
pretty = PrettyPrinter(indent = 2)

pretty.pprint(results)
```

Take a moment to familiarse with this data structure, all our responses from DBpedia will be in a similar format.

Now we'll use the FILTER constraint to filter out all responses other than those in English and Spanish.

```
endpoint.setQuery('''
```

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpr: <http://dbpedia.org/resource/>
SELECT ?label
WHERE {
  dbpr:Asturias rdfs:label ?label .
  FILTER( LANG(?label)="es" || LANG(?label)="en") .
}
''')
```

As before, we'll execute the query and print the results:

```
results = endpoint.query().convert()

for result in results['results']['bindings']:
  print(result['label']['value'])
```

We didn't need to specify the endpoint url or return format because we already did that previously. You can read more about the FILTER constraint here: https://jena.apache.org/tutorials/sparql_filters.html.

Try changing the query to select responses in some languages that aren't English or Spanish. (Hint, run the query with no FILTER and look at the `xml:lang` predicate).

3. Using SPARQLWrapper and rdflib together

Now we'll use our knowledge from last week to create an RDF graph and populate it with the result from some SPARQL queries.

For clarity, let's import everything we need first (even though some of these things are already imported).

```
import rdflib
from rdflib import plugin
from rdflib.graph import Graph
from rdflib.namespace import Namespace
from SPARQLWrapper import SPARQLWrapper, JSON, XML

endpoint = SPARQLWrapper('http://dbpedia.org/sparql')
```

We'll be using a CONSTRUCT query which tells SPARQL to construct new rdf triples based on the result of the query. For simplicity we'll simply create an exact copy of the graph.

```
query = '''
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
CONSTRUCT {
    <%(uri)s> a ?type .
    <%(uri)s> ?property ?value .
    <%(uri)s> rdfs:label ?label .
    ?value rdfs:label ?vlabel .
    ?property rdfs:label ?plabel .
}
WHERE {
    <%(uri)s> a ?type.
    <%(uri)s> ?property ?value .
    <%(uri)s> rdfs:label ?label .
    ?value rdfs:label ?vlabel .
    ?property rdfs:label ?plabel .

    FILTER( LANG(?label)="es" || LANG(?label)="en" ) .
}
'''
```

The above query string is using a Python feature called *named placeholders*. You can read a bit about them here: https://www.dummies.com/programming/python/use-named-arguments-in-format-strings/

We use the `%` operator to replace the named placeholders with another string, it saves us some typing!

```
endpoint.setQuery(query % { 'uri': 'http://dbpedia.org/resource/Asturias' })
```

This time we'll set the return format to XML:

```
endpoint.setReturnFormat(XML)
```

And then execute the query (note: this may take a little time, don't worry!)

```
graph = endpoint.query().convert()

for s, p, o in graph:
  print('Subject => ' + s)
  print('Predicate => ' + p)
  print('Object => ' + o)
```

There might be a few errors, this is OK we can ignore them.

The `graph` variable is an rdflib graph like the ones we were creating and manipulating last week. Today we'll use a new method, `graph.query` to write SPARQL queries for our local rdf graph.

```
query = '''
PREFIX dbpo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?location ?party ?leader
WHERE {
    <%(uri)s> rdfs:label ?location .
    <%(uri)s> dbpo:leaderName ?leaderResource .
    ?leaderResource rdfs:label ?leader .
    <%(uri)s> dbpo:leaderParty ?partyResource .
    ?partyResource rdfs:label ?party .
    FILTER( LANG(?location)="%(lang)s" && LANG(?party)="%(lang)s" && LANG(?lea
}
'''

replacements = {
  'uri': 'http://dbpedia.org/resource/Asturias',
  'lang': 'es'
}
```

Now we can run the query just like we did for our SPARQL endpoint:

```
results = graph.query(query % replacements)

for row in results:
  print('The leading party of %s is %s. Their leader is %s.' % row)
```

4. The Interactive DBpedia Endpoint

DBpedia provides a web-based tool for writing and testing SPARQL queries. You can access it here: http://dbpedia.org/isparql/

On the Advanced tab, type or copy the following query into the text area:

```
PREFIX dbpedia: <http://dbpedia.org/resource/>
```

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpo: <http://dbpedia.org/ontology/>
SELECT DISTINCT ?location ?leader ?party
FROM <http://dbpedia.org>
WHERE {
  dbpedia:Asturias rdfs:label ?location ;
  dbpo:leaderName ?leaderResource .
  ?leaderResource rdfs:label ?leader .
  dbpedia:Asturias dbpo:leaderParty ?partyResource .
  ?partyResource rdfs:label ?party .
}
```

You can then click the play button to run the query and explore the results.

You may also navigate to the QBE tab (query by example) and click the "Get from Advanced" icon (it looks like a sheet of paper with a green download arrow). This will generate a visualisation of the query that can be handy to see what it is you're selecting.

Some errors may pop up when loading the query from Advanced, we can ignore those.