Introduction to Python and RDFlib

> **Session objectives:** - Become familiar with using Python to create simple Semantic Web programs. - Learn how to use RDFlib to load and manipulate RDF graphs.

1. Introduction

In this lab we'll be using Python to develop simple scripts that access, query, and manipulate RDF semantic data.

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:

```
python 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:

```
python
```

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/

2. The RDFLib Graph

We'll be working with the serialised RDF graph file `shakespeare.n3`. If it was not included with this document, you can download it yourself from: https://raw.githubusercontent.com/QMUL-ECS735P/lab-week-3/master/shakespeare.n3

Open the file in a text editor and identify each RDF triple; how many are there? Have a look at both the prefixes and the defined triples. How many different concepts are defined in this RDF graph? How many properties?

Before we can run python on the ITL machines, we need to set up the python environment by entering the following in the terminal:

```
module load python/3.6.6
```

This step as crucial to ensure the machines load the correct version of python and the libraries we depend on. You will have to do this every week!

If you type now `python` in the terminal you should see the following output:

```
bash-4.2$ python
Python 3.6.6 (default, Sep  3 2018, 15:31:46)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Take a moment to familiarse yourself with python and the interactive shell by running some of the commands details here: https://docs.python.org/3/tutorial/introduction.html#using-python-as-a-calculator

---

Before using functions and classes from RDFlib, we need to add some import declarations. Evaluate the following code line-by-line:

```
from __future__ import print_function, unicode_literals
import warnings
warnings.filterwarnings('ignore')

import rdflib
from rdflib.graph import Graph, Store, URIRef, BNode, Literal
from rdflib.namespace import Namespace, RDF, RDFS
from rdflib import plugin
```

The Graph class in RDFlib represents a set of triples. We're going to create a new Graph and populate it with the definitions in `shakespeare.n3`.

```
g = Graph()
g.parse('shakespeare.n3', format = 'n3')
```

If `shakespeare.n3` is saved in a different folder, make sure you write the correct path to the file. We can also create the graph by fetching the data directly from the internet:

```
g.parse('https://raw.githubusercontent.com/QMUL-ECS735P/lab-week-3/master/shak
```

The RDF graph is now loaded in memory. We can check the number of statements using the `len` function:

```
len(g)
```

Or an internal representation of the object using `repr`:

```
repr(g)
```

Now, use a `for` loop to iterate through the contents of the graph (you can find the documentation for loops here: http://docs.python.org/tutorial/controlflow.html#for-statements). For now we'll just print each statement out.

> Python is whitespace sensitive! We don't use `{ ... }` to denote blocks of code, instead we use indentation such as spaces or tabs.

```
for st in g:
  print(st)
```

We can also use *destructuring* to unpack the subject, predicate, and object automatically in the loop:

```
for s, p, o in g:
  print("subject: " + str(s))
  print("predicate: " + str(p))
```

```
  print("object is: " + str(o))
```

The actual elements in the triples are Python classes that represent URIRefs, blank nodes, and literals. Write a loop that uses the `type` function to print the type of each variable such as `type(s)`.

You can read more about the Graph class here: https://rdflib.readthedocs.io/en/6.1.1/intro_to_graphs.html

3. Serialisation

With our shakespeare graph still stored in the `g` variable, we can use the `serialize` method to see what the graph looks like in different formats:

```
print(g.serialize(format = 'nt'))
print(g.serialize(format = 'turtle'))
print(g.serialize(format = 'xml'))
```

Compare each format, note how the `turtle` format is much less verbose than N-triples (`nt`) or `xml`. Read up how to save files using python and save each serialisation to a new file such as `shakespeare.xml`. Here is a good place to start: https://www.w3schools.com/python/python_file_write.asp

4. The RDF Store

For this part of the lab, we'll be using RDFlib to create an RDF Store. Stores allow us to persist our graphs in muliple ways such as a SQL database. For simplicity, we'll be using an in-memory store: in your own time you might be interested in exploring how to use MySQL, SQLite, or Sleepycat.

We'll create some example RDF data to manipulate with. For reference, you can find that example data here: https://www.w3schools.com/xml/xml_rdf.asp.

```
rdf_xml_data = '''<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:cd="htt
<rdf:Description
rdf:about="http://www.recshop.fake/cd/Empire_Burlesque">
  <cd:artist>Bob_Dylan</cd:artist>
  <cd:country>USA</cd:country>
  <cd:company>Columbia</cd:company>
  <cd:price>10.90</cd:price>
  <cd:year>1985</cd:year>
</rdf:Description>
<rdf:Description
rdf:about="http://www.recshop.fake/cd/Hide_your_heart">
  <cd:artist>Bonnie_Tyler</cd:artist>
  <cd:country>UK</cd:country>
  <cd:company>CBS_Records</cd:company>
  <cd:price>9.90</cd:price>
  <cd:year>1988</cd:year>
</rdf:Description>
</rdf:RDF>
'''
```

The three apostrophes (`'''`) indicate a multiline string.

With our example data ready, we can begin by creating an empty RDF store:

```
memory_store = plugin.get('Memory', Store)()
```

Stores need a base URI to link concepts to, we'll use a fake example URI for this exercise:

```
graph_id = URIRef('http://example.com/foo')
```

Finally, and as before, we'll create an RDF graph but this time we'll use the store and id defined above:

```
g = Graph(store = memory_store, identifier = graph_id)
```

If you closed it, open back up the `shakespeare.n3` file in a text editor. Have a look at some of the properties defined, such as `married` or `partOf`. We're going to define some new triples using these properties.

What makes the Semantic Web powerful is the ability to reference and use concepts defined in other graphs to create a web of shared meaning. This is achieved through URI namespacing.

RDFlib predefines the RDF and RDFS namespaces, but we're going to add two more from the `shakespeare.n3` file:

```
nslit = Namespace('http://www.workingontologist.org/Examples/Chapter3/shakespe
nsbio = Namespace('http://www.workingontologist.org/Examples/Chapter3/biograph
g.bind('lit',nslit)
g.bind('bio', nsbio)
```

These namespaces correspond to the following namespaces defined in `shakespeare.n3`:

```
@prefix lit:     <http://www.workingontologist.org/Examples/Chapter3/shakespea
@prefix bio:     <http://www.workingontologist.org/Examples/Chapter3/biography
```

We can check al the bound prefixes in our current graph with another `for` loop:

```
for (p, n) in g.namespaces():
  print("Prefix: " + str(p) + ". Corresponds to namespace: " + str(n))
```

Now we have our namespaces defined, we can start adding new RDF triples to the graph. To do this we'll use the `g.add` method:

```
g.add( (nsbio['Cervantes'], RDF.type, nsbio['Person']) )
g.add( (nsbio['Cervantes'], RDFS.label, Literal('Viguel_de_Cervantes')) )
g.add( (URIRef('http://example.com/bar'), RDFS.label, Literal('bar')) )
```

Look back at Section 2 and use another `for` loop to print out our new RDF triples. Then look at Section 3 and print the graph serialised in the `turtle` format.

You might notice that some parts are neatly prefixed such as `rdfs:label` but `<http://example.com/bar>` is not. This is because of the prefix binding we did earlier.

Let's bind a new prefix `ex` to clear up our graph:

```
g.bind('ex', 'http://example.com')
```

Printing the serialised graph again we can see `bar` is now properly abbreviated.

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

Similarly, printing the namespaces again confirms a new namespace has been added.

```
for (p, n) in g.namespaces():
  print("Prefix: " + str(p) + ". Corresponds to namespace: " + str(n))
```

Now we're going to take the xml data we prepared at the start of this step and add it to our RDF graph.

Begin by printing the number of triples currently in the graph using the `len` function:

```
print('Number of triples in the graph: %i' %len(g))
```

The above print looks a bit different to what we've done previously. This is known as string interpolation, the result of `len(g)` is automatically converted to a string and inserted where `%i` is in the string.

We should have 3 triples in the graph.

Now, parse the xml data and store it in our RDF graph:

```
g.parse(data = rdf_xml_data, format = 'application/rdf+xml')
```

We can confirm the xml data was properly parsed by checking the number of triples now in the graph. We should now have 13 triples.

The `g.objects` method allows us to query the graph and receive all objects that match the supplied subject and/or predicate. You can read about this method (and others) here: https://rdflib.readthedocs.io/en/4.2.2/intro_to_graphs.html

Let's use it to print all music artists in our graph:

```
artists = g.objects(subject = None, predicate = URIRef('http://www.recshop.fak
for artist in artists:
    print(artist)
```

Use everything you've learned so far to add some new data to the graph by creating a new CD and artist. Experiment with the `g.subjects`, `g.objects`, and `g.predicates` methods to get familiar with how to query the graph.

5. Visualising RDF Graphs

There are a number of websites that allow us to visualise our graphs so we don't have to simply read plain text. These websites use a `.dot` file containing the tuples to produce the visualisation, so we will need to write a python function to convert our graph to this format.

```
def triplesToDot (triples, filename, nsdict):
  out = open(filename, 'w')
  out.write('graph "SimpleGraph" {\n')
  out.write('overlap = "scale";\n')
  for t in triples:
    s = '"%s" -- "%s " [label="%s"] ;\n' % (t[0].encode('utf-8'), t[2].encode(
    for item in nsdict:
      s = s.replace(item, nsdict[item])
    out.write(s)
  out.write('}')
```

Don't worry if you don't quite understand what's happening here, as you become more familiar with python it will be clearer.

Next we're going to create a python dictionary of all the namespaces in our graph, a dictionary is a collection

of key/value pairs.

```
namespaces = {}
for (p, n) in g.namespaces():
  print(n, p + ':')
  namespaces[n] = p + ':'
```

Finally we'll call the `triplesToDot` function to create a `.dot` file from our graph:

```
triplesToDot(g, 'week-3.dot', namespaces)
```

Open `week-3.dot` in a text editor, copy the text and paste it in one of these websites: - http://www.webgraphviz.com/ - http://viz-js.com/ - https://dreampuf.github.io/GraphvizOnline/

These tools can be handy to get a clearer idea of what is happening when you change a graph. Continue using what you've learned today to manipulate your RDF graph in python. Use the `triplesToDot` function to visualise the changes you're making.

6. Extra Practice

When you are finished, head on over to the RDFlib documentation here: https://rdflib.readthedocs.io/en/6.1.1/. Tasks under the **Getting started** heading cover the same material that we have covered today, so checking there is a good place if you're stuck or unclear of anything. To get a deeper understanding of how RDFlib works, check out some of the links under the **In depth** heading.

For an even more detailed resource, the O'Reilly book "Programming the Semantic Web" covers much of the material we will cover both in the labs and lectures and would serve as a useful reference throughout the module.