2021/2022 2Q

Documentación de Laboratorio

ECSDI

Javier Béjar

Departament de Ciències de la Computació

Grau en Enginyeria Informàtica - UPC

UPC FIB Facultat d'Informàtica de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

FACULTAT D'INFORMÁTICA DE BARCELONA
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Primera edición, Febrero 2014

Esta edición, Febrero 2022

Índice general

Capítulo 1

Python

Python[1] es un lenguaje creado por Guido van Rossum pensado como un lenguaje sencillo y fácil de aprender.

Es un lenguaje con tipado dinámico, con varias estructuras de datos de alto nivel como parte propia del lenguaje (listas, diccionarios) y una aproximación sencilla a la orientación a objetos. Tiene también elementos de programación funcional y muchas extensiones que permiten acceder a otros paradigmas de programación.

Su sintaxis es sencilla, prescindiendo de declaración de variables y parentizaciones de bloque, permitiendo un código más comparto y legible que el de otros lenguajes de programación.

El gran número de librerias disponibles sobre prácticamente cualquier dominio de aplicación le permiten ser el lenguaje de preferencia para muchos desarrolladores.

En la asignatura usaremos Python 3, podéis encontrar documentación introductoria en:

- https://docs.python.org/3/tutorial/
- http://en.wikibooks.org/wiki/A_Beginner's_Python_Tutorial
- http://www.openbookproject.net/pybiblio/
- Y obviamente en la web de Python http://www.python.org/

1.1. Instalación de paquetes y librerías

En la red de PCs tendréis instalados todos los paquetes que necesitáis en linux. Para los paquetes que puedan faltar o paquetes adicionales que queráis instalar los tendréis que instalar localmente. Podéis hacer esto usando el comando `pip`.

Para hacerlo con `pip` basta con hacer `pip install --user <paquete>` se os instalará el paquete en vuestra zona de usuario (bajo el directorio `~/.local`).

También podéis utilizar la libreria `env` o `virtualenv` para crear una versión local de python sobre la que podéis trabajar sin tener que mezclar paquetes con la distribución que tenéis instalada en los linux de la red de PCs (podéis usar `pip` para instalarlos).

Por ejemplo, teniendo instalado `virtualenv`, haciendo:

```
virtualenv --system-site-packages /directorio/destino
```

[1] Nombre en honor a los Monty Python.

crearéis un entorno python en el directorio que le indiquéis (con el `-system-site-packages` permitimos que también se tenga acceso a los paquetes instalados en el sistema, evitando el tener que reinstalar cosas). Para activar el entorno python deberéis moveros al directorio de instalación y ejecutar:

```
source bin/activate
```

a partir de ese momento el ejecutable de python pasará a ser el del entorno virtual, sabréis que ha funcionado si el prompt del terminal cambia al nombre del entorno virtual. Para desactivarlo solo tenéis que hacer:

```
deactivate
```

Cuando activéis el entorno virtual podréis instalar paquetes con `pip`.

El entorno virtual tendrá los paquetes del sistema, así que tendréis que instalar solo los paquetes adicionales que queráis usar para desarrollar la práctica.

Los paquetes que serán necesarios para la práctica son:

- `Multiprocessing`, librería para programación multiproceso y concurrente
- `flask` y `flask-restful`, librerías para la creación de servicios web RESTful
- `requests`, librería para hacer peticiones a servicios web RESTful
- `rdflib`, librería para ontologías y web semántica
- `SPARQLWrapper`, librería para hacer consultas a SPARQL points (se instala al instalar `rdflib`)

Cuando empecéis a desarrollar la practica también le tendréis que indicar a python donde se encuentran las librerías que tendrá vuestro código (y las que tenéis disponibles en el github de la asignatura). Para ello tendréis que definir la variable de entorno `PYTHONPATH` para que apunte a esos directorios. Para que sea más sencillo podéis definirla en el fichero de configuración de vuestro shell (`.tcshrc`, `.bashrc`...).

Capítulo 2

Concurrencia en Python

2.1. Introducción

Un agente está compuesto de diferentes comportamientos que se ejecutan de manera concurrente. Estos comportamientos procesan las percepciones del agente, reaccionan a estas percepciones y toman decisiones a partir del estado del agente y sus objetivos.

Para simular los diferentes comportamientos del agente utilizaremos diferentes hilos concurrentes que se encargarán de implementarlos.

2.2. Multiprocessing

La librería `Multiprocessing` permite una implementación sencilla de hilos concurrentes. Se puede encontrar su documentación en http://docs.python.org/2/library/multiprocessing.html. Esta librería dispone de todos los elementos necesarios para implementar un programa que necesita ejecutar varios procesos a la vez, además de elementos de comunicación entre los procesos y estructuras compartidas.

2.3. Procesos

Se puede crear un nuevo proceso en un programa utilizando la clase `Process`. El contructor de la clase recibe una función y los argumentos de la función como los parámetros `target` y `args`. Hay que tener en cuenta que el proceso recibe una copia de los argumentos que se le pasan, así que si queremos que una estructura esté compartida entre varios procesos debemos utilizar las facilidades de esta librería que lo permiten.

En este ejemplo creamos dos subprocesos que ejecutan la función `cuenta`, que recibe dos parámetros indicando los límites entre los que tiene que contar e imprime los números. Los dos subprocesos se ejecutan a la vez y el proceso principal los inicia y espera a que terminen.

```
1 from multiprocessing import Process
2
3 def cuenta(li,ls):
4     for i in range(li,ls):
5       print i,'\n'
6
```

```
if __name__ == '__main__':
    p1 = Process(target=cuenta, args=(10,20,))
    p2 = Process(target=cuenta, args=(20,30,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

2.4. Comunicación entre procesos

Los procesos creados pueden comunicarse entre si usando colas (clase `Queue`) y tuberias (clase `Pipe`).

La clase `Queue` es una cola compartida en la que se pueden insertar y sacar objetos. Todos los procesos que comparten la cola pueden poner objetos y consumirlos. Para poner objetos usamos `put` y para consumirlos `get`, podemos bloquearnos hasta que la operación tenga éxito y establecer un tiempo de espera.

En este ejemplo el proceso principal manda los numeros de 0 al 9 a través de la cola y el subproceso los va imprimiendo.

```
from multiprocessing import Process, Queue
import time

def cuenta(q):
    time.sleep(1)
    while q.empty():
        pass
    while not q.empty():
        print q.get(timeout=0.3)
        time.sleep(1)

if __name__ == '__main__':
    q=Queue()
    p = Process(target=cuenta, args=(q,))
    p.start()
    for i in range(10):
        q.put(i)
    p.join()
```

La clase `Pipe` es un canal de comunicación que puede ser bidireccional (por defecto) o no. Este objeto retorna un par de objetos de la clase `Connection` a partir de los cuales se realiza la comunicación. El envío de mensajes se realiza mediante la función `send` y la recepción mediante la función `recv`, podemos consultar si hay objetos en la conexión mediante la función `poll` y cerrar la conexión mediante la función `close`.

En este ejemplo dos procesos reciben los dos extremos de una tubería y se van intercambiando los numeros del 1 al 9 e imprimiéndolos.

```
from multiprocessing import Process, Pipe

def proceso1(conn):
    for i in range(10):
        conn.send(i)
```

```
6           print conn.recv(), 'P1:'
7       conn.close()
8
9   def proceso2(conn):
10      for i in range(10):
11          print conn.recv(),'P2:'
12          conn.send(i)
13      conn.close()
14
15  if __name__ == '__main__':
16      conn1, conn2 = Pipe()
17      p1 = Process(target=proceso1, args=(conn1,))
18      p1.start()
19      p2 = Process(target=proceso2, args=(conn2,))
20      p2.start()
21      p1.join()
22      p2.join()
```

2.5. Primitivas de sincronización

Esta librería dispone también de diferentes clases que implementan las primitivas de sincronización habituales en programación concurrente (semáforos, eventos, condiciones y cerrojos).

El objeto más sencillo es el cerrojo `Lock` que podemos cerrar (`acquire`) y abrir (`release`).

Lo podemos utilizar para asegurar un acceso exclusivo a un recurso. Por ejemplo una estructura de datos compartida.

Este ejemplo hace lo mismo que el anterior, pero asegura que cuando un proceso está escribiendo en el vector, el otro no lo hace.

```
1   from multiprocessing import Process, Array, Lock
2   from ctypes import c_int
3
4   def proceso1(a,l):
5      l.acquire()
6      print a[:]
7      for i in range(0,10,2):
8           a[i] = i*i
9      l.release()
10
11  def proceso2(a,l):
12     l.acquire()
13     print a[:]
14     for i in range(1,10,2):
15          a[i] = i*i
16     l.release()
17
18  if __name__ == '__main__':
19      arr = Array(c_int, 10)
20      l = Lock()
```

```
    p1 = Process(target=proceso1, args=(arr,1,))
    p1.start()
    p2 = Process(target=proceso2, args=(arr,1,))
    p2.start()
    p1.join()
    p2.join()

    print arr[:]
```

2.6. Estado compartido

Esta librería tiene dos objetos simples que permiten compartir estado entre múltiples procesos `Value` y `Array`.

El objeto `Value` recibe dos parámetros, el tipo y el valor inicial. El objeto `Array` recibe el tipo y el tamaño o un inicializador. El tipo se puede indicar usando los tipos definidos en la librería `ctypes`. Podemos proteger el acceso a los datos, si es necesario, mediante un cerrojo pasando como parámetro `lock` el valor `True` (esto generará un cerrojo) o pasar uno nosotros como parámetro (clase `Lock`).

En este ejemplo dos procesos van poniendo los valores de los cuadrados del 0 al 9 en un vector en las posiciones pares e impares.

```
from multiprocessing import Process, Array
from ctypes import c_int

def proceso1(a):
   for i in range(0,10,2):
        a[i] = i*i

def proceso2(a):
   for i in range(1,10,2):
        a[i] = i*i

if __name__ == '__main__':
    arr = Array(c_int, 10)

    p1 = Process(target=proceso1, args=(arr,))
    p1.start()
    p2 = Process(target=proceso2, args=(arr,))
    p2.start()
    p1.join()
    p2.join()

    print arr[:]
```

Si queremos compartir estructuras más complejas debemos utilizar la clase `Manager`. Implementa un proceso que permite el acceso a información compartida a otros procesos. Este proceso puede estar en una máquina distinta a los procesos con los que se comparte las estructuras.

Con esta clase se pueden generar, entre otras estructuras, listas (`list`) y diccionarios (`dict`) que se pueden compartir. También se pueden crear espacios de nombre compartidos (`Namespace`) que permiten

asignarles diferentes identificadores que son visibles entre los procesos. A estos identificadores les podemos asignar estructuras de datos.

La particularidad de estos espacios y estructuras compartidas, es que los objetos mutables que les asignemos no propagaran automáticamente sus modificaciones entre los procesos. Esto quiere decir que si asignamos a un espacio de nombres una estructura compleja o tenemos en una lista/diccionario un objeto no primitivo (otra lista por ejemplo) y la cambiamos, no veremos las modificaciones a no ser que la volvamos a reasignar al espacio de nombres o lista/diccionario.

Esto también quiere decir que si dos procesos pueden escribir a la vez en la estructura, deberemos utilizar alguna primitiva de sincronización para que unas modificaciones no reescriban otras.

En este ejemplo dos procesos comparten un diccionario a través de un `Namespace` de un `Manager` y escriben en el diccionario. Primero obtienen la estructura, escriben en ella y luego la vuelven a colocar en el `Namespace`, para que los cambios se propaguen. Para evitar que una modificación machaque a la otra utilizamos un cerrojo que cerramos al adquirir el diccionario y liberamos al retornarlo al espacio de nombres.

```
1  from multiprocessing import Process, Manager, Lock
2
3  def proceso1(nsp, l):
4     l.acquire()
5     data = nsp.data
6     a = ['a','b','c']
7     for i,v in enumerate(a):
8         data[v] =  i
9     nsp.data = data
10    l.release()
11
12 def proceso2(nsp, l):
13    l.acquire()
14    data = nsp.data
15    a = ['e','f','g']
16    for i,v in enumerate(a):
17        data[v] = i
18    nsp.data = data
19    l.release()
20
21 if __name__ == '__main__':
22     shnsp = Manager().Namespace()
23     l = Lock()
24
25     shnsp.data={}
26
27     p1 = Process(target=proceso1, args=(shnsp,l,))
28     p2 = Process(target=proceso2, args=(shnsp,l,))
29     p1.start()
30     p2.start()
31     p1.join()
32     p2.join()
33
34     print shnsp.data
```

Capítulo 3

WebServices RESTful

3.1. Introducción

La arquitectura de servicios web basada en REST (REpresentational State Transfer) permite crear APIs web sencillas que pueden utilizarse para modelar todo tipo de aplicaciones. En este caso la usaremos como metodología de desarrollo de sistemas multiagentes desplegados en un entorno distribuido.

Los servicios REST no estan estandarizados como por ejemplo SOAP, pero su simplicidad permite una mayor flexibilidad a la hora de desarrollar. Aun así, pueden utilizar estándares por ejemplo para el intercambio de información como XML.

Un servicio esta asociado a un recurso que tiene una direccion (URL, URI) a partir de la cual se puede hacer la comunicación. Un recurso está formado por un conjunto de objetos (entendiendo por objeto cualquier tipo de información) que es accesible/modificable a través de sus operaciones.

El énfasis de REST está en la simplicidad. Se definen un conjunto de operaciones (verbos) que son comunes a todas las aplicaciones/APIs. Estas operaciones son las operaciones estándar que todo servidor web es capaz de procesar:

- GET: Retorna toda la información del recurso (sin efectos laterales) o la correspondiente a la URI enviada (o acorde con los parámetros en la petición)
- PUT: Substituye toda información del recurso por la que se envía o la del item identificado por la URI.
- POST: Crea una nueva entrada en el recurso recibiendo la URI asignada.
- DELETE: Elimina toda la información del recurso o de la URI indicada

El intercambio de información puede realizarse mediante cualquier formato, pero lo usual es el uso de estándares basados en XML o más recientemente usando el formato JSON.

3.2. Flask

Flask es un framework implementado en python que permite crear APIs web de manera sencilla.

Podéis encontrar esta API en http://flask.pocoo.org/. Ahí tenéis toda la documentación detallada y múltiples ejemplos. Tenéis también un tutorial sencillo en http://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask. Este documento explicará lo mínimo para crear un servicio web usando esta API.

Sobre Flask se ha desarrollado una extensión que simplifica algunas tareas en el desarrollo de servicios web REST. Esta extensión se llama Flask-RESTful, podéis encontrar su documentación en http://flask-restful.readthedocs.org/en/latest/.

3.2.1. Servicios REST en Flask

Un servicio REST debe definir un conjunto de puntos de entrada (recursos) que sirven a las diferentes acciones del servicio y las distintas operaciones REST. Una aplicación de Flask se implementa a partir de la clase `Flask`. Con un objeto de esta clase podemos definir las rutas donde se encontrarán las funciones de la API que se pueden llamar.

Una ruta se define con el decorador `@app` de la siguiente manera `@app.route('/path/to/operation')`. Por ejemplo, esta sería una aplicación mínima que responde a invocaciones `GET` en el path raíz (`/`) con el consabido `"Hello, World!"`

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/')
6  def hello():
7      return "Hello, World!"
8
9  if __name__ == '__main__':
10     app.run()
```

Si ejecutamos este programa (`app.py`) obtendremos:

```
$ python app.py
 * Running on http://127.0.0.1:5000/
```

Y podemos invocar el método usando `curl`:

```
$ curl http://127.0.0.1:5000/
127.0.0.1 - - [20/Jan/2014 16:35:32] "GET / HTTP/1.1" 200 -
Hello, World!
```

Por defecto la aplicación se pone en marcha en la dirección local (`127.0.0.1`) y en el puerto 5000, pero podemos configurarla mediante los parámetros `host` y `port` de la siguiente manera

```
1  app.run(host='miweb.org',port=9999)
```

Podemos definir los métodos que admitimos en una ruta especificándolo en el parámetro `methods` que tiene `app.route`. El objeto *request* tiene el atributo `methods` que permite acceder al tipo de petición recibida, por ejemplo:

```
1  @app.route('/agente', methods=['GET', 'POST'])
2  def peticion_objeto():
3      if request.method == 'GET':
4          return 'Hola, soy un agente'
5      else:
6          return 'POST recibido'
```

El objeto `request` almacena toda la información referente a la petición (mirad la documentación para una descripción completa), entre ellas los argumentos de la llamada (`request.args`) y los campos enviados (`request.form`) y los ficheros que se han enviado (`request.files`) en un `POST`. Todos estos elementos se almacenan como un diccionario python. Por ejemplo, la función siguiente extrae los argumentos `x` e `y` de la llamada y los retorna sumados:

```
1  @app.route('/sumador')
2  def sumador():
3      x = int(request.args['x'])
4      y = int(request.args['y'])
5      return str(x+y)
```

3.3. Requests

La librería Requests permite hacer peticiones a un servidor web de manera sencilla, podéis encontrar su documentación en http://docs.python-requests.org/en/latest/.

Básicamente, esta librería encapsula los distintos tipos de operaciones (GET, PUT, POST, DELETE), permitiendo generar las llamadas y procesar las respuestas.

Esta clase dispone de los métodos básicos que nos hacen falta para invocar servicios:

- `requests.get('url')`
- `requests.post('url')`
- `requests.put('url')`
- `requests.delete('url')`

Los parámetros se pasan a las peticiones como un diccionario python, por ejemplo:

```
1  peticion = {'usuario': 'pepe', 'password:': '1234'}
2  r= requests.get('http://unsecure.site.com',params=peticion)
```

Podemos tener acceso a la respuesta que hemos obtenido del servidor a partir de los diferentes atributos del objeto que recibimos con la petición. Por ejemplo, el atributo `text` nos dará la respuesta como una cadena, el atributo `content` nos la dará en formato binario.

Si estamos accediendo a un servicio que nos retorna datos en formato JSON, podemos decodificarlo usando el método `json()`.

Capítulo 4

Web Semántica

4.1. Introducción

La Web Semántica es una iniciativa promovida por el W3C que entre otras cosas tiene como objetivo el anotar semánticamente el contenido de la WWW de manera que pueda ser también procesado de manera automática.

Esta iniciativa incluye una serie de estándares que permiten representar el contenido de los recursos de la WWW a partir de ontologías, obtener información de los diferentes recursos de la web semántica a partir de un lenguaje de consultas (SPARQL) y hacer razonamiento sobre el contenido de la web semántica.

4.2. rdflib

La librería rdflib permite crear, manipular, consultar y almacenar grafos RDF (y OWL). Se puede encontrar la documentación en `https://rdflib.readthedocs.org/en/latest/`.

4.2.1. Grafos, nodos, literales, tripletas

La estructura básica de la librería es el objeto `Graph`, que permitirá almacenar las definiciones y hechos que representan un dominio.

Un grafo estará compuesto por tripletas (triplets) compuestas por sujeto, predicado y objeto. Estos elementos podrán ser nodos (con un URI) o literales.

Podemos crear un nodo con una URI específica usando la clase `URIRef`, nodos vacíos para los que se generará una URI mediante la clase **BNode** o literales mediante la clase `Literal`, por ejemplo:

```
1 from rdflib import URIRef, BNode, Literal
2
3 pedro = URIRef('http://mundo.mundial.org/persona/pedro')
4 maria = BNode()
5
6 nombre = Literal('Pedro')
7 edad = Literal(22)
```

Podemos definir un espacio de nombres y crear nodos dentro de este espacio mediante la clase `Namespace`. Por ejemplo:

```
from rdflib import Namespace

myns = Namespace('http://my.namespace.org/personas')

a = myns.tomas
#rdflib.term.URIRef(u'http://my.namespace.org/personas/tomas'
```

Hay definidas clases que representan espacios de nombre usuales como `RDF` y `FOAF`.

Podemos añadir tripletas a un grafo mediante el método `add`, por ejemplo:

```
from rdflib import URIRef, BNode, Literal, Namespace, RDF, FOAF

g = Graph()

mm = Namespace('http://mundo.mundial.org/persona/')

pedro = mm.pedro
maria = mm.maria

g.add((pedro, RDF.type, FOAF.persona))
g.add((maria, RDF.type, FOAF.persona))
g.add((pedro, FOAF.name, Literal('Pedro')))
g.add((maria, FOAF.name, Literal('Maria')))
g.add((pedro, FOAF.knows, maria))
```

Podemos modificar el valor de una propiedad funcional (cardinalidad 1) mediante el método `set`:

```
g.add((pedro, FOAF.age, Literal(22)))

g.set((pedro, FOAF.age, Literal(23)))
```

Podemos eliminar una tripleta mediante el método `remove`, indicando la tripleta específica o usando `None` para dejar sin especificar algún elemento.

```
g.add((pedro, RDF.type, FOAF.persona))
g.add((maria, RDF.type, FOAF.persona))

g.add((pedro, FOAF.age, Literal(22)))
g.add((maria, FOAF.age, Literal(23)))

# Eliminar la edad de pedro
g.remove((pedro, FOAF.age, Literal(22)))

# Eliminar todas las tripletas que se refieren a maria
g.remove((maria, None, None))
```

4.2.2. Cargando y grabando ficheros RDF

Los ficheros RDF contendrán la información en una de las múltiples formas de serializarlos, lo que nos interesa es convertirlos a un grafo para poder trabajar con su contenido. Para cargar un fichero lo primero

que deberemos saber es en que formato está serializado (xml, n3, triplets...) y utilizar el método `parse` del objeto `Graph`, por ejemplo:

```
g= Graph()
g.parse('http://my.ontology.org/ontologia.owl',format='xml')
```

cargará el fichero `owl` referenciado con la url. Se puede cargar un fichero local, o usar una URL para cargar ficheros remotos.

Para grabar los datos que tenemos en un grafo RDF podemos usar el método `serialize`, por ejemplo:

```
g = Graph()
n = Namespace('http://ejemplo.org/')

p1 = n.persona1
v = Literal(22)
g.add((p1, FOAF.age, v))
g.serialize('a.rdf')
```

creará un grafo que tendrá una tripleta indicando la edad de una persona y lo grabará en el fichero `a.rdf` en el formato por defecto (XML).

4.2.3. Consultas sobre un grafo

La clase `Graph` permite diferentes formas por las que se puede consultar su contenido. En primer lugar se puede iterar sobre una variable de este tipo obteniendo todas las tripletas que contiene, por ejemplo:

```
g = Graph()
n = Namespace('http://ejemplo.org/')

p1 = n.persona1
v = Literal(22)
g.add((p1, FOAF.age, v))

for s, p, o in g
  print s, p, o
```

retornaría los valores de la tripleta que hemos almacenado en el grafo. También podemos seleccionar tripletas del grafo a partir de un sujeto específico, por ejemplo:

```
for p, o in g[p1]
  print p, o
```

retornaría los predicados y objetos que están relacionados con el sujeto `p1`.

También tenemos las siguientes funciones para hacer consultas:

- `objects`, retorna los objetos relacionados con el sujeto y predicado que se pasa como parámetro.
- `predicates`, retorna los predicados relacionados con el sujeto y objeto que se pasa como parámetro.

- `subjects`, retorna los sujetos relacionados con el predicado y objeto que se pasa como parámetro.
- `predicate_objects`, retorna los predicados y objetos relacionados con el sujeto que se pasa como parámetro.
- `subject_objects`, retorna los sujetos y objetos relacionados con el predicado se pasa como parámetro.
- `subject_predicates`, retorna los sujetos y predicados relacionados con el objeto que se pasa como parámetro.

También podemos usar SPARQL para hacer consultas sobre el grafo usando `query`, que recibe como parámetro un string con la consulta. Finalmente, podemos usar también el método `triples` para hacer una consulta simple, poniendo `None` en la parte de la tripleta que queremos que sea variable, por ejemplo:

```
g.triples((None, FOAF.age, Literal(22)))
```

retornará todas las tripletas que tengan 22 años de edad.

El operador `in` esta sobrecargado para los grafos, así que podemos hacer consultas sencillas de existencia con esta sintaxis:

```
if (mm.pedro, RDF.type, FOAF.persons) in g:
    print "Pedro es una persona"
```

4.2.4. Uso de SPARQL

El acceso a consultas mas complejas y a modificaciones en el grafo RDF se puede realizar utilizando el lenguaje SPARQL mendiante los métodos `query` y `update`. Por ejemplo:

```
res = g.query("""
          PREFIX foaf: <http://xmlns.com/foaf/0.1/>
          SELECT DISTINCT ?a ?name
          WHERE {
          ?a foaf:age ?edad .
          ?a foaf:name ?name .
          FILTER (?edad > 18)
        }
       """)
```

Retornaría una lista de personas con sus nombres que tienen más de 18 años. Podemos recorrer esa lista para acceder a los resultados.

Este método tiene un parámetro `initNs` al que se le puede pasar un diccionario con espacios de nombre para no tener que añadir la definición de los prefijos usados, por ejemplo:

```
res = g.query("""
          SELECT DISTINCT ?a ?name
          WHERE {
          ?a foaf:age ?edad .
          ?a foaf:name ?name .
          FILTER (?edad > 18)
        }
       """, initNs = {'foaf', FOAF})
```

Tendría el mismo resultado (FOAF esta ya definido en RDFLib).

Si queremos hacer modificaciones en el grafo podemos hacer por ejemplo

```
g.update("""
          PREFIX foaf: <http://xmlns.com/foaf/0.1/>
          INSERT DATA
          {
            juan a foaf:person;
                 foaf:name "Juan".
          }
      """)
```

Añade al grafo una nueva persona (juan) con su nombre.

4.3. SPARQLWrapper

SPARQLWrapper es una clase de python que facilita el hacer consultas a un SPARQL point. Al crear un objeto de la clase se le debe pasar la URL que tiene el SPARQL point. A partir de ese momento se pueden mandar consultas en SPARQL a ese objeto que se encarga de hacer la consulta y recibir los resultados. Por defecto retorna los resultados como un grafo RDF, pero no siempre puede convertirse a un grafo válido, por lo que es mas seguro utilizar el formato JSON.

El siguiente código hace una consulta a DBPedia, obteniendo un objeto JSON que es después transformado a diccionarios python. Los diccionarios guardan la estructura del resultado, básicamente las vinculaciones de las variables de la query, podemos acceder a ellas dentro de la clave del diccionario `["results"]["bindings"]`.

```
from SPARQLWrapper import SPARQLWrapper, JSON

sparql = SPARQLWrapper("http://dbpedia.org/sparql")
sparql.setQuery("""
  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
  SELECT ?label
  WHERE { <http://dbpedia.org/resource/Asturias> rdfs:label ?label }
""")
sparql.setReturnFormat(JSON)
results = sparql.query().convert()

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

Si lo que queremos es obtener un grafo RDF con la query, podemos utilizar la consulta CONSTRUCT de SPARQL, usando el formato RDF nos retorna un grafo de RDFLib, por ejemplo:

```
sparql = SPARQLWrapper("http://dbpedia.org/sparql")
sparql.setQuery("""
  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
  CONSTRUCT {<http://dbpedia.org/resource/Asturias> rdfs:label ?label}
  WHERE { <http://dbpedia.org/resource/Asturias> rdfs:label ?label }
""")
```

```
7  sparql.setReturnFormat(RDF)
8  grafo = sparql.query().convert()
9
10 for _, _, o in grafo:
11   print o
```

retornaría:

```
Asturies
Asturien
Astúrias
Asturië (regio)
Asturie
Asturias
Asturia
Asturias
```

Capítulo 5

Fuentes de información

5.1. Ontologías

Ya existen diferentes ontologías sobre productos que podéis usar para escribir la vuestra. Por ejemplo:

1. `schema.org` (https://schema.org/docs/schemas.html) tiene una clase para productos y algunas más específicas para elementos concretos, como por ejemplo libros.
2. Good Relations (http://www.heppnetz.de/ontologies/goodrelations/v1.html) es específica para e-commerce e incluye muchas clases de productos.

En la web de Open Linked Vocabularies (https://lov.linkeddata.es/dataset/lov/) podéis encontrar otros vocabularios.

5.2. Datos de productos

En el dominio de la práctica es difícil obtener bases de datos o servicios gratuitos o de libre acceso con información.

Obviamente, la mejor fuente sería `amazon.com` pero no ofrece la informacion de sus productos de manera gratuita y también es muy estricta respecto a recolectar datos de su web mediante bots.

La principal utilidad de sus páginas es para definir la ontología de productos a usar en la práctica, tanto la categorización como los diferentes campos que se utilizan para describir los productos. Obviamente, no hace falta que incluyáis todo lo que hay.

Una posibilidad es que generéis de manera aleatoria una base de datos de productos. Una vez definida la ontología no es difícil el hacer un programa que vaya generando instancias de las diferentes clases dando valores aleatorios a los diferentes campos dentro de los rangos que tengan sentido. En el repositorio de código dentro de la carpeta `Examples/InfoSources` tenéis el script `RandomInfo.py` que os puede servir de ejemplo.

Capítulo 6

Agentes ejemplo

En el directorio AgentExamples del repositorio tenéis tres ejemplos de agentes básicos que os pueden ayudar a la hora de ver como se implementa un agente y como se hace la comunicación.

6.1. SimpleDirectoryService

Este agente implementa un servicio de directorio en el que agentes que proveen servicios se pueden registrar y en los que otros agentes pueden buscar. Este agente no mantiene la persistencia del directorio.

La implementación utiliza Flask para permitir la comunicación mediante REST. Hay tres rutas que implementan el comportamiento:

- `/Register`, que permite hacer el registro y la búsqueda en el directorio
- `/Info`, que muestra una página web con las peticiones de registro que se han recibido (en formato turtle) que se puede ver por ejemplo conectándose al agente con un navegador
- `/Stop`, que para el agente

Las dos últimas rutas no requieren muchas explicaciones. La primera es la que se encarga de recibir y procesar los mensajes de registro y búsqueda.

El proceso del mensaje tiene que seguir el protocolo de peticiones entre agentes. Este caso es simple, si recibimos un mensaje que no corresponde con lo que esperamos debemos contestar diciendo que no lo hemos entendido, si es una petición de registro debemos confirmar la acción y si es de búsqueda hemos de informar del resultado.

El proceso que sigue la implementación es el siguiente, el mensaje inicial se recibe en el parámetro de contenido de la llamada a esta entrada de la API, que sera un mensaje RDF/OWL serializado en XML. Este mensaje sigue el formato de FIPA-ACL. El vocabulario de esta ontología se importa de la clase `AgentUtil.ACL` como un `ClosedNamespace`, de manera que no se pueda comprobar que se usa correctamente el vocabulario. Para poder manipularlo hay que convertirlo en un grafo RDF, para ello solo tenemos que hacer un parsing del contenido a una variable de tipo grafo RDF.

Para facilitar el tratamiento del mensaje la función `ACLMessages.get_message_properties` extrae los campos del mensaje como un diccionario a partir del cual se pueden obtener sus valores.

En este punto tenemos que decidir que parte del protocolo hemos de ejecutar. Si la extracción de los campos del mensaje no ha tenido éxito, es que el mensaje no es comprensible y retornamos un mensaje

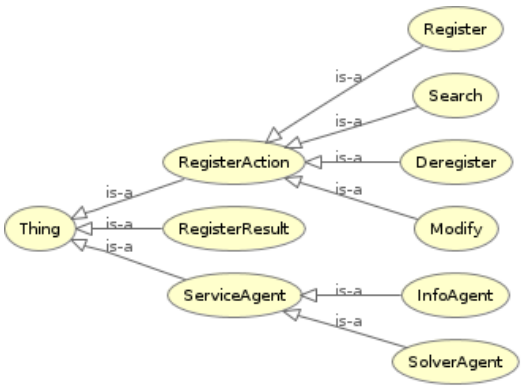

Figura 6.1: Esquema de la ontología Directory Service

FIPA-ACL con performativa `not-understood`. La función `ACLMessages.build_message` se encarga de construir el mensaje FIPA-ACL a partir de sus elementos.

Si el mensaje es válido, hay que comprobar que tenga la performativa correcta, en este caso ha de ser del tipo `request`. Si no es así, también respondemos con un `not-understood`. Si es un request hemos de obtener del contenido el tipo de acción que se solicita.

Las acciones posibles estan definidas en la ontología Directory Service Ontology (ver 6.1) que encontraréis en el directorio `Ontologias` del repositorio. Estas se importan de la clase `AgentUtil.DSO` que las define como un `ClosedNamespace` de RDFLib. Esto permite que se compruebe que no se usan términos de la ontología que no están en el vocabulario. Este agente solo implementa el proceso de dos acciones `DSO.Register` y `DSO.Search`.

Si es una acción de registro se extraen los elementos necesarios para registrar al agente del mensaje (Address, Name, Uri, AgentType) y se insertan en la variable global `dsgraph` que contiene un grafo RDF que guarda la información de registro de los agentes. Esta información se representa usando DSO y FOAF (ontología Friend of a Friend) de la que se usa la clase `FOAF.Agent` que es la que se asocia a la URI del agente y la relación `FOAF.Name`, para el resto de campos se utiliza DSO. Estos son los datos mínimos que se pueden usar para registrar un agente, se puede extender DSO o usar otros elementos de FOAF (u otras ontologías) para añadir más información al registro. El protocolo de comunicación en este caso se finaliza enviando un mensaje con performativa `confirm` para informar al agente de que el registro se ha realizado.

Esta claro que hay otros casos que no se contemplan en este protocolo y que se deberían tratar para tener un servicio de registro más robusto, como por ejemplo el caso de que el agente ya esté registrado, o que haya algún error en los datos del agente.

Si es una acción de búsqueda la implementación solo considera la posibilidad de buscar por tipo de agente. Esta información es uno de los parámetros de esta acción en la ontología. Podemos extraer esta información del contenido y hacer una consulta al grafo RDF que contiene los registros. Únicamente se retorna el primer agente encontrado, buscando su dirección a partir de la URI que se obtiene de la búsqueda. Con esta información se envía como respuesta un objeto de tipo `DSO.Directory-response` que tiene como parámetros la dirección del agente y su URI. Esta respuesta es enviada como un mensaje FIPA-ACL con performativa `inform`.

El resto de acciones que tiene la ontología Directory Service no están implementadas, pero la forma de hacerlo sería similar.

Adicionalmente a las entradas de la API que gestiona Flask veréis que se crea un proceso (`agentbehavior1`) que permite que el agente haga otras cosas de manera concurrente mientras está esperando mensajes. En este caso este proceso recibe una cola para poder comunicarse con otros procesos del agente. Esta función en realidad lo único que hace es esperar que llegue información a la cola y acaba cuando llega un cero. Obviamente, podemos implementar tareas más complejas y tener varias tareas concurrentes haciendo cosas.

6.2. SimpleInfoAgent

Este agente es un esqueleto de agente de información/resolución que tiene también tres rutas:

- `/comm`, que es el comportamiento que procesa las peticiones que recibe
- `/iface`, que muestra una pagina web con un contenido fijo
- `/Stop`, que para el agente

Al inicio, el agente pone en marcha un proceso concurrente que hace el registro del agente en el agente de directorio (asume que tiene una dirección fija conocida) y se queda esperando leyendo de la cola que recibe como parámetro. Este proceso acaba cuando recibe un 0 a través de la cola. El registro del agente se hace generando un mensaje FIPA-ACL con performativa `request` que tiene como contenido una acción de registro de la ontología DSO con los datos del agente. Para el envío del mensaje se utiliza la función `AgentUtil.send_message` que se encarga de serializar el mensaje, enviarlo a la dirección del agente destino, esperar la respuesta y convertirla en un grafo RDF.

La entrada `/comm` esta a la espera de peticiones. El protocolo de interacción es sencillo, los mensajes deben tener la performativa `request` y deberán de ser acciones que pertenezcan a la ontología que haya definido el agente (u otra general para este tipo de agentes). La implementación responde un `not-understood` si el mensaje no es valido o no es un `request`, en otro caso extrae la acción del contenido y la realiza, respondiendo con el resultado. En la implementación concreta no se realiza ningún procesado de la acción y se responde siempre con un mensaje con performativa `inform-done`.

6.3. SimplePersonalAgent

Este agente es un esqueleto de agente que busca y utiliza a otros agentes para resolver problemas, tiene tres rutas

- `/comm`, que es el comportamiento que procesa las peticiones que recibe (no hace nada)
- `/iface`, que muestra una página web con un contenido fijo
- `/Stop`, que para el agente

Todo el comportamiento esta implementado en el proceso concurrente que se pone en marcha al principio del agente, pero obviamente se puede utilizar la ruta de comunicación para hacer cosas más complejas. Este comportamiento hace una secuencia de acciones fija que emplea a los dos agentes anteriores. Primero hace una búsqueda en el agente de directorio enviando una acción del tipo `DSO.Search` para encontrar agentes del tipo `DSO.HotelAgent` (en la ontología DSO hay definidos unos pocos tipos de agentes). Del mensaje recibido extrae la información necesaria para contactar al agente que se ha encontrado y le hace una petición que pertenece a la ontología `IAActions` (no está definida). Una vez recibida la respuesta el agente acaba su ejecución.

6.4. Ejecución de los agentes

Los agentes de ejemplo está implementados de manera que ejecuten las acciones que se han descrito. Primero se ha de poner en marcha `SimpleDirectoryService` que espera las peticiones de registro. Después se ha de poner en marcha `SimpleInfoAgent` que se registra en el directorio y queda a la espera de

peticiones. Por último se ha de poner en marcha `SimplePersonalAgent` que hace la búsqueda en el servicio de directorio y hace una petición al agente de información. Los agentes se conectan a los puertos 9000, 9001 y 9002 de la máquina local (`127.0.0.1`). Se puede acceder a las entradas de los agentes que muestran una página de web desde un navegador y a la entrada que para a los agentes.

Si queremos que la comunicación entre los agentes se haga desde maquinas distintas hay que cambiar el parámetro `host` del método `run` de la aplicacion Flask para que sea '0.0.0.0'. A partir de ahí el servidor web acepta conexiones remotas. En este caso habrá que indicar a cada agente la dirección IP (o hostname) que tienen los otros agentes con los que ha de comunicarse.