2021/2022 2Q

Documentazione di laboratorio

Javier Béjar

Dipartimento di Scienze della Calcolazione

Grau in Ingegneria Informatica - UPC

FACOLTÀ DI INFORMATICA DI BARCELLONA
UNIVERSITÀ POLITICA DI CATALONIA

Prima edizione, febbraio 2014

Questa edizione, febbraio 2022

Indice generale

Capitolo 1

Pitone

Python[1] è un linguaggio creato da Guido van Rossum come linguaggio semplice e di facile apr comprensione.

È un linguaggio tipizzato dinamicamente, con diverse strutture dati di alto livello come parte del da linguaggio stesso (liste, dizionari) e un semplice approccio orientato agli oggetti. Ha anche elementi di programmazione funzionale e molte estensioni che permettono di accedere ad altri paradigmi di programmazione.

programmazione.

La sua sintassi è semplice e rinuncia alle dichiarazioni di variabili e al parsing dei blocchi, consentendo un codice più condiviso e leggibile rispetto ad altri linguaggi di programmazione.

L'elevato numero di librerie disponibili praticamente in ogni ambito applicativo lo rende il linguaggio ess preferito da molti sviluppatori.

ere Nel corso utilizzeremo Python 3; la documentazione introduttiva è disponibile all'indirizzo:

- https://docs.python.org/3/tutorial/
- http://en.wikibooks.org/wiki/A_Beginner's_Python_Tutorial
- http://www.openbookproject.net/pybiblio/
- E naturalmente sul sito web di Python http://www.python.org/

1.1. Installazione di pacchetti e librerie

Sulla rete del PC saranno installati tutti i pacchetti necessari in linux. I pacchetti mancanti o aggiuntivi che si desidera installare, devono essere installati localmente. È possibile farlo utilizzando il comando pip.

Per farlo con pip, basta pip install --user <pacchetto> e il pacchetto sarà installato nella vostra area utente (sotto la directory ~/.local).

Si può anche usare la libreria env o virtualenv per creare una versione locale di python su cui che lavorare senza dover mischiare i pacchetti con la distribuzione installata su linux del PC (si può usare ret pip per installarli).

e Ad esempio, avendo installato virtualenv, facendo:

```
virtualenv --system-site-packages /directory/target
```

[1] Prende il nome dai Monty Python.

creerà un ambiente python nella directory specificata (con -system-site-packages si avrà accesso anche ai pacchetti installati sul sistema, evitando di doverli reinstallare). Per attivare l'ambiente python è necessario spostarsi nella directory di installazione ed eseguire:

```
sorgente bin/activate
```

Da quel momento in poi, l'eseguibile python diventerà l'eseguibile dell'ambiente virtuale; saprete che ha funzionato se il prompt del terminale cambierà nel nome dell'ambiente virtuale. Per disattivarlo è sufficiente farlo:

```
disattivare
```

Una volta attivato l'ambiente virtuale, sarà possibile installare i pacchetti con pip.

L'ambiente virtuale avrà i pacchetti di sistema, quindi dovrete installare solo i pacchetti aggiuntivi che volete usare per sviluppare la pratica.

I pacchetti che saranno necessari per la pratica sono:

- Multiprocessing, libreria per il multiprocessing e la programmazione
- concorrente flask e flask-restful, librerie per la creazione di servizi web
- RESTful Richieste RESTful, libreria per effettuare richieste a servizi web

  RESTful
- rdflib, libreria per le ontologie e il web semantico
- SPARQLWrapper, libreria per l'interrogazione di punti SPARQL (installata con l'installazione di rdflib).

Quando iniziate a sviluppare la pratica, dovrete anche dire a python dove si trovano le librerie che il vostro codice avrà (e quelle che avete a disposizione nel github del corso). Per farlo, occorre impostare la variabile d'ambiente PYTHONPATH in modo che punti a queste directory. Per semplificare le cose, potete definirlo nel file di configurazione della vostra shell (.tcshrc, .bashrc...).

Capitolo 2

Concorrenza in Python

2.1. Introduzione

Un agente è composto da diversi comportamenti che vengono eseguiti simultaneamente. Questi comportamenti elaborano le percezioni dell'agente, reagiscono a queste percezioni e prendono decisioni basate sullo stato e sugli obiettivi dell'agente.

Per simulare i diversi comportamenti degli agenti utilizzeremo diversi thread concorrenti che avrà il a compito di implementare.

2.2. Multiprocesso

La libreria Multiprocessing consente una semplice implementazione di thread concorrenti. La documentazione è disponibile all'indirizzo http://docs.python.org/2/library/multiprocessing.html. Questa libreria contiene tutti gli elementi necessari per implementare un programma che deve eseguire diversi processi contemporaneamente, nonché elementi di comunicazione tra i processi e strutture condivise.

2.3. Processi

Un nuovo processo può essere creato in un programma utilizzando la classe Process. Il costruttore della classe riceve una funzione e gli argomenti della funzione come parametri target e args. Si noti che il processo riceve una copia degli argomenti che gli sono stati passati, quindi se si vuole che una struttura sia condivisa tra più processi, bisogna utilizzare le funzioni di questa libreria che lo consentono.

In questo esempio creiamo due thread che eseguono la funzione count, che riceve due parametri che indicano i limiti tra i quali deve contare e stampa i numeri. I due thread vengono eseguiti contemporaneamente e il processo principale li avvia e attende che finiscano.

```
1da multiprocessing importa Processo

3def conto(li,ls):
4per     i in range(li,ls):
5print      i,'\n'
```

```
7se nome_____== ' principale ':
8p1     = Process(target=conto, args=(10,20,))
9p2     = Process(target=conto, args=(20,30,))
10      p1.start()
11      p2.start()
12      p1.join()
13      p2.join()
```

2.4. Comunicazione tra processi

I processi creati possono comunicare tra loro utilizzando code (classe Queue) e pipe (classe Pipe).

La classe Queue è una coda condivisa in cui gli oggetti possono essere inseriti e rimossi. Tutti i processi che condividono la coda possono inserire oggetti e consumarli. Per inserire gli oggetti si usa put e per consumarli si usa get, si può bloccare finché l'operazione non ha successo e impostare un timeout.

In questo esempio, il processo principale invia i numeri da 0 a 9 attraverso la coda e il sottoprocesso li stampa.

```
1from multiprocessing import Process, Queue
2tempo di importazione

4def conto(q):
5       tempo. sleep(1)
6       while q. empty():
7pass
8       mentre non q. empty():
9           print q. get(timeout=0,3)
10          tempo. sleep(1)

12se nome_____== ' principale ':
13q=Queue    ()
14p     = Process(target=account, args=(q,))
15      p. start()
16per   i in range(10):
17          q. mettere(i)
18      p. join()
```

La classe Pipe è un canale di comunicazione che può essere bidirezionale (per impostazione predefinita) o meno. Questo oggetto restituisce una coppia di oggetti della classe Connection da cui avviene la comunicazione. I messaggi vengono inviati con la funzione send e ricevuti con la funzione recv. Possiamo controllare se ci sono oggetti nella connessione con la funzione poll e chiudere la connessione con la funzione close.

In questo esempio, due processi ricevono le due estremità di un tubo e scambiano i numeri da 1 a 9 e li stampano.

```
1da multiprocessing importare Processo, Tubo

3def processo1(conn):
4per    i in range(10):
5           conn. invia(i)
```

```
6               print conn. recv(), 'P1:'
7       conn. close()
8
9def process2(conn):
10per       i in range(10):
11              print conn. recv(),'P2:'
12              conn. invia(i)
13      conn. close()

15if  nome_____== '  principale ':
16conn1      , conn2 = Pipe()
17p1         = Processo(target=processo1, args=(conn1,))
18          p1.start()
19p2         = Processo(target=processo2, args=(conn2,))
20          p2.start()
21          p1.join()
22          p2.join()
```

2.5. Primitive di sincronizzazione

Questa libreria ha anche diverse classi che implementano le consuete primitive di sincronizzazione nella programmazione concorrente (semafori, eventi, condizioni e latches).

L'oggetto più semplice è il blocco, che possiamo acquisire e rilasciare.

Può essere utilizzato per garantire l'accesso esclusivo a una risorsa. Ad esempio, una struttura dati condivisa.

Questo esempio fa la stessa cosa del precedente, ma assicura che quando un processo scrive sul vettore, l'altro non lo fa.

```
1from  multiprocessing import Process, Array, Lock
2from  ctypes import c_int

4def processo1(a,l):
5l          . acquisire()
6stampa     a[:]
7per        i in range(0,10,2):
8a              [i] = i*i
9l          . rilascio()

11def process2(a,l):
12l         . acquisire()
13print      a[:]
14per       i in range(1,10,2):
15a             [i] = i*i
16l         . rilascio()

18if  nome_____== '  principale ':
19arr       = Array(c_int, 10)
20l         = Blocco()
```

```
21        p1 = Processo(target=processo1, args=(arr,l,))
          p1.start()
          p2 = Processo(target=processo2, args=(arr,l,))
          p2.start()
25        p1.join()
26        p2.join()

          stampare arr[:]
```

2.6. Stato condiviso

Questa libreria ha due semplici oggetti che permettono di condividere lo stato tra più processi Value e Array.

L'oggetto Value riceve due parametri, il tipo e il valore iniziale. L'oggetto Array riceve il tipo e la dimensione o un inizializzatore. Il tipo può essere specificato utilizzando i tipi definiti nella libreria ctypes. Possiamo proteggere l'accesso ai dati, se necessario, tramite un blocco, passando come parametro lock il valore True (questo genererà un blocco) o passando un nostro parametro (classe Lock).

In questo esempio, due processi inseriscono i valori dei quadrati da 0 a 9 in un vettore nelle posizioni pari e dispari.

```
1from  multiprocessing import Process, Array
2from  ctypes import c_int

4def processo1(a):
5per      i in range(0,10,2):
6a              [i] = i*i

8def processo2(a):
9per      i in range(1,10,2):
10a             [i] = i*i

12se  nome_____== '  principale ':
13arr       = Array(c_int, 10)

15p1        = Processo(target=processo1, args=(arr,))
16          p1.start()
17p2         = Processo(target=processo2, args=(arr,))
18          p2.start()
19          p1.join()
20          p2.join()
21
22print      arr[:]
```

Se vogliamo condividere strutture più complesse, dobbiamo usare la classe Manager. Implementa un processo che consente l'accesso alle informazioni condivise da altri processi. Questo processo può trovarsi su una macchina diversa dai processi con cui le strutture sono condivise.

Con questa classe è possibile generare, tra le altre strutture, elenchi (list) e dizionari (dict) che possono essere condivisi. È anche possibile creare spazi dei nomi condivisi (Namespace) che permettono di

assegnare loro identificatori diversi, visibili tra i processi. A questi identificatori possiamo assegnare strutture di dati.

La particolarità di questi spazi e strutture condivisi è che gli oggetti mutabili devi assegniamo non propagheranno automaticamente le loro modifiche tra i processi. Ciò significa che se assegniamo una struttura complessa a uno spazio dei nomi o se abbiamo un oggetto non primitivo in un elenco/dizionario (un altro elenco, per esempio) e lo cambiamo, non vedremo le modifiche a meno che non lo riassegniamo allo spazio dei nomi o all'elenco/dizionario.

Questo significa anche che se due processi possono scrivere sulla struttura nello stesso momento, dobbiamo usare qualche primitiva di sincronizzazione in modo che alcune modifiche non ne riscrivano altre.

In questo esempio, due processi condividono un dizionario attraverso un Manager Namespace e scrivono sul dizionario. Prima ottengono la struttura, la scrivono e poi la rimettono nel Namespace, in modo che le modifiche si propaghino. Per evitare che una modifica sovrascriva un'altra, utilizziamo un blocco che blocchiamo quando acquisiamo il dizionario e rilasciamo quando lo restituiamo allo spazio dei nomi.

```
1da multiprocessing import Processo, Manager, Blocco

3def processo1(nsp, l):
4l      . acquisire()
5       dati = dati nsp.
6a      = ['a', 'b', 'c']
7per    i,v in enumerate(a):
8data       [v] =       i
9       nsp. dati = dati
10l     . rilascio()

12def  process2(nsp, l):
13l     . acquisire()
14      dati = dati nsp.
15a     = ['e', 'f', 'g']
16per   i,v in enumerate(a):
17dati      [v] = i
18      nsp. dati = dati
19l     . rilascio()

21if nome_____== '  principale ':
22      shnsp = Manager(). Spazio dei nomi()
23l     = Blocco()

25      shnsp. data={}
26
27p1     = Processo(target=processo1, args=(shnsp,l,))
28p2     = Processo(target=processo2, args=(shnsp,l,))
29      p1.start()
30      p2.start()
31      p1.join()
32      p2.join()

34      stampare i dati di shnsp.
```

Capitolo 3

Servizi Web REST

3.1. Introduzione

L'architettura dei servizi web basata su REST (REpresentational State Transfer) consente di creare wsemplici API che possono essere utilizzate per modellare tutti i tipi di applicazioni. In questo caso lo utilizzeremo come
metodologia di sviluppo per sistemi multi-agente distribuiti.

I servizi REST non sono standardizzati come, ad esempio, SOAP, ma la loro semplicità consente una unatmaggiore flessibilità nello sviluppo. Anche in questo caso, possono comunque utilizzare gli standard, ad esempio per il
scambio di informazioni come XML.

Un servizio è associato a una risorsa che ha un indirizzo (URL, URI) da cui può avvenire la cer comunicazione. Una risorsa è costituita da un insieme di oggetti (definiti come qualsiasi oggetto). tipo di informazione) che è accessibile/modificabile attraverso le sue operazioni.

L'enfasi di REST è sulla semplicità. Viene definito un insieme di operazioni (verbi) comuni a tutte le applicazioni/API. Queste operazioni sono quelle standard che ogni server web è in grado di elaborare:

- GET: Restituisce tutte le informazioni sulla risorsa (senza effetti collaterali) o quelle corrispondenti all'URI inviato (o in base ai parametri della richiesta).
- PUT: Sostituisce tutte le informazioni sulla risorsa con quelle inviate o con quelle dell'elemento
- identificato dall'URI. POST: Crea una nuova voce nella risorsa ricevendo l'URI assegnato.
- DELETE: Cancella tutte le informazioni dalla risorsa o dall'URI specificato.

Lo scambio di informazioni può avvenire in qualsiasi formato, ma è consuetudine utilizzare gli eststandard di basati su XML o, più recentemente, sul formato JSON.

3.2. Fiaschetta

Flask è un framework implementato in python che consente di creare API web in modo semplice.

Questa API è disponibile all'indirizzo http://flask.pocoo.org/. Qui trovate tutta la documentazione y dettagliata e numerosi esempi. Si può anche trovare un semplice tutorial su http://blog.miguelgrinberg.com/post/ designing-a-restful-api-with-python-and-flask. Questo documento illustra i requisiti minimi per creare un servizio Web utilizzando questa API.

È stata sviluppata un'estensione di Flask che semplifica alcuni compiti nello sviluppo di servizi web RESTful. Questa estensione si chiama Flask-RESTful e la sua documentazione è disponibile all'indirizzo http:
//flask-restful.readthedocs.org/it/latest/.

3.2.1. Servizi REST in Flask

Un servizio REST deve definire un insieme di punti di ingresso (risorse) che servono alle diverse azioni del servizio e alle diverse operazioni REST. Un'applicazione Flask è implementata dalla classe Flask. Con un oggetto di questa classe si possono definire le rotte in cui si trovano le funzioni API che possono essere richiamate.

Una rotta è definita con il decoratore @app come segue @app. route('/path/to/operation'). Ad esempio, si tratterebbe di un'applicazione minimale che risponde alle invocazioni GET sul percorso radice (/) con il familiare "Hello, World!".

```
1from flask import Flask

3app = Flask( nome )

5 @app. route('/')
6def hello():
7     restituire "Hello, World!"
8
9se nome_____== ' principale ':
10     app. run()
```

Se eseguiamo questo programma (app.py) otterremo:

```
$ python app.py
 * Esecuzione su http://127.0.0.1:5000/
```

E possiamo invocare il metodo usando curl:

```
$ curl http://127.0.0.1:5000/
127.0.0.1 - - - [20/gennaio/2014 16:35:32] "GET / HTTP/1.1" 200 -
Ciao, mondo!
```

Per impostazione predefinita, l'applicazione viene avviata sull'indirizzo locale (127.0.0.1) e sulla porta 5000, ma è possibile configurarla utilizzando i parametri host e porta come segue

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

Possiamo definire i metodi che supportiamo in una rotta, specificandoli nel parametro methods di app.route. L'oggetto *request* ha un attributo methods che consente di accedere al tipo di richiesta ricevuta, ad esempio:

```
1  @app. route('/agent', methods=['GET', 'POST'])
2def object_request():
3       se richiesta. metodo == 'GET':
4Ritornare          "Salve, sono un agente".
5else       :
6ritorno            'POST ricevuto
```

L'oggetto request memorizza tutte le informazioni relative alla richiesta (vedere la documentazione per una descrizione completa), compresi gli argomenti della chiamata (request.args) e i campi inviati (request.form) e i file che sono stati inviati (request.files) in un POST. Tutti questi elementi sono memorizzati in un dizionario python. Ad esempio, la funzione seguente estrae gli argomenti x e y dalla chiamata e li restituisce sommati:

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

3.3. Richieste

La libreria Requests consente di effettuare richieste a un server web in modo semplice; la sua documentazione è disponibile all'indirizzo http://docs.python-requests.org/en/latest/.

In sostanza, questa libreria incapsula i diversi tipi di operazioni (GET, PUT, POST, DELETE), consentendo di generare le chiamate e di elaborare le risposte.

Questa classe contiene i metodi di base necessari per invocare i servizi:

- richieste. get('url')
- richieste. post('url')
- richieste. put('url')
- richieste.

  delete('url')

I parametri vengono passati alle richieste come un dizionario python, ad esempio:

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

Possiamo accedere alla risposta ottenuta dal server dai diversi attributi dell'oggetto ricevuto con la richiesta. Ad esempio, l'attributo text fornirà la risposta come stringa, mentre l'attributo content fornirà la risposta in formato binario.

Se si accede a un servizio che restituisce dati in formato JSON, è possibile decodificarli utilizzando il metodo json().

Capitolo 4

Web semantico

4.1. Introduzione

Il Web semantico è un'iniziativa promossa dal W3C che mira, tra l'altro, ad annotare semanticamente i s contenuti del WWW in modo che possano essere elaborati in modo semantico.
utomatico.

Questa iniziativa comprende una serie di standard che permettono di rappresentare il contenuto delle il risorse WWW sulla base di ontologie, di ottenere informazioni sulle diverse risorse del web semantico dada un linguaggio di interrogazione (SPARQL) e di ragionare sul contenuto del web semantico.

4.2. rdflib

La libreria rdflib consente di creare, manipolare, interrogare e memorizzare grafi RDF (e OWL). La documentazione è disponibile all'indirizzo o https://rdflib.readthedocs.org/en/latest/.

4.2.1. Grafi, nodi, letterali, triple

La struttura di base della libreria è l'oggetto Graph, che consente di memorizzare le definizioni e i fatti che che rappresentano un dominio.

Un grafo è composto da triplette costituite da soggetto, predicato e oggetto. Questi elementi possono essere nodi (con un URI) o letterali.

Possiamo creare un nodo con un URI specifico usando la classe URIRef, nodi vuoti per i quali sarà generato un URI usando la classe **BNode** o letterali usando la classe Literal, per esempio:

```
1from rdflib import URIRef, BNode, Literal

3pedro = URIRef('http://mundo.mundial.org/persona/pedro')
4maria = BNode()
5
6nome = Letterale('Pedro')
7age = Letterale(22)
```

Possiamo definire uno spazio dei nomi e creare nodi all'interno di questo spazio dei nomi utilizzando la classe Da Namespace. esempio:

```
from rdflib import Namespace

myns = Spazio dei nomi('http://my.namespace.org/personas')

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

Esistono classi definite che rappresentano spazi dei nomi comuni, come RDF e FOAF. Possiamo aggiungere triple a una rete usando il metodo add, ad esempio:

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

g = Grafico()

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

pedro = mm. pedro
maria = mm. maria

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

Possiamo modificare il valore di una proprietà funzionale (cardinalità 1) utilizzando il metodo set:

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

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

Possiamo rimuovere una tripletta utilizzando il metodo remove, specificando la tripletta specifica o utilizzando il metodo
Nessuno per lasciare un elemento non specificato.

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

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

# Rimuovere l'età di Pedro
g. remove((pedro, FOAF. age, Literal(22)))

# Rimuovere tutte le triplette che si riferiscono a Maria
g. remove((maria, Nessuno, Nessuno, Nessuno))
```

4.2.2. Caricamento e salvataggio di file RDF

I file RDF conterranno le informazioni in uno dei tanti modi di serializzazione, quello che ci interessa è convertirli in un grafo per poter lavorare con il loro contenuto. Per caricare un file, la prima cosa da fare è

dobbiamo sapere in quale formato è serializzato (xml, n3, triplette.) e utilizzare il metodo parse della classe Oggetto grafico, ad esempio:

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

caricherà il file owl a cui fa riferimento l'url. È possibile caricare un file locale o utilizzare un URL per caricare file remoti.

Per salvare i dati che abbiamo in un grafo RDF, possiamo usare il metodo serialize, ad esempio:

```
g = Grafico()
n = Spazio dei nomi('http://ejemplo.org/')

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

creare un grafo che abbia una tripletta che indichi l'età di una persona e salvarlo nel file a.rdf nel formato predefinito (XML).

4.2.3. Interrogazione in rete

La classe Graph consente di interrogare il suo contenuto in diversi modi. In primo luogo, è possibile iterare su una variabile di questo tipo per ottenere tutte le triple che contiene, ad esempio:

```
g = Grafico()
n = Spazio dei nomi('http://ejemplo.org/')

p1 = n. persona1
v = Letterale(22)
g. add((p1, FOAF. età, v))

per s, p, o in g
stampa s, p, o
```

restituirebbe i valori delle triple memorizzate nella rete. Possiamo anche selezionare le triple della rete da un argomento specifico, ad esempio:

```
per p, o in g[p1]
stampa p, o
```

restituirebbe i predicati e gli oggetti relativi al soggetto p1. Sono disponibili

anche le seguenti funzioni per effettuare interrogazioni:

- restituisce gli oggetti relativi al soggetto e al predicato passati come parametro.
- restituisce i predicati relativi al soggetto e all'oggetto passati come parametro.

- restituisce i soggetti relativi al predicato e all'oggetto passati come parametro.
- predicate_objects, restituisce i predicati e gli oggetti relativi all'argomento passato come parametro.
- subject_objects, restituisce i soggetti e gli oggetti relativi al predicato passato come parametro.
- subject_predicates, restituisce i soggetti e i predicati relativi all'oggetto passato come parametro.

Possiamo anche usare SPARQL per fare delle interrogazioni sul grafo, usando query, che riceve come parametro una stringa con l'interrogazione. Infine, possiamo anche usare il metodo delle triple per fare una semplice query, inserendo None nella parte della tripla che vogliamo sia variabile, ad esempio:

```
1 g. triple((Nessuno, FOAF. età, Letterale(22))))
```

restituirà tutte le triple che hanno 22 anni.

L'operatore in è sovraccaricato per i grafi, quindi possiamo fare semplici interrogazioni sull'esistenza con questa sintassi:

```
1 se (mm. pedro, RDF. tipo, FOAF. persone) in g:
2Stampa "Pietro è una persona".
```

4.2.4. Uso di SPARQL

L'accesso a interrogazioni più complesse e le modifiche al grafo RDF possono essere eseguite utilizzando il linguaggio SPARQL attraverso i metodi di interrogazione e aggiornamento. Ad esempio:

```
1 res = g. query("""")
2PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3SELEZIONA DISTINTO ?a ?nome
4DOVE
5 ?a foaf:age ?age .
6 ?a foaf:name ?name .
7FILTRO (?età > 18)
8}
9 """)
```

Il programma restituisce un elenco di persone con i loro nomi che hanno più di 18 anni. È possibile scorrere l'elenco per accedere ai risultati.

Questo metodo ha un parametro initNs al quale si può passare un dizionario con gli spazi dei nomi, in modo da non dover aggiungere la definizione dei prefissi utilizzati, ad esempio:

```
1res = g. query("""")
 SELEZIONA DISTINTO ?a
 ?nome
 DOVE {
 .a foaf:age .age .
5 .a foaf:name .name .
 FILTRO (?età > 18)
7}
8 """, initNs = {'foaf', FOAF})
```

Avrebbe lo stesso risultato (FOAF è già definito in RDFLib).

Se si vogliono apportare modifiche alla rete, si può fare, ad esempio, quanto segue

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

Aggiungere una nuova persona (John) alla rete con il suo nome.

4.3. SPARQLWrapper

SPARQLWrapper è una classe python che facilita l'interrogazione di un punto SPARQL. Quando si crea un oggetto della classe, è necessario passargli l'URL del punto SPARQL. Da quel momento in poi è possibile inviare query SPARQL a quell'oggetto, che si occupa di effettuare la query e di ricevere i risultati. Per impostazione predefinita, restituisce i risultati come un grafo RDF, ma non sempre è possibile convertirlo in un grafo valido, quindi è più sicuro usare il formato JSON.

Il codice seguente interroga DBPedia, ottenendo un oggetto JSON che viene poi trasformato in dizionari python. I dizionari memorizzano la struttura del risultato, fondamentalmente i binding
delle variabili della query, possiamo accedervi all'interno del dizionario chiave ["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#>
SELEZIONA ?etichetta
WHERE { <http://dbpedia.org/resource/Asturias> rdfs:label ?label }
""")
sparql.setReturnFormat(JSON)
risultati = sparql.query().convert()

per risultato in risultati["risultati"]["vincoli"]:
print(result["label"]["value"])
```

Se vogliamo ottenere un grafo RDF con la query, possiamo usare la query CONSTRUCT di SPARQL, utilizzando il formato RDF per restituire un grafo RDFLib, ad esempio:

```
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 }
""")
```

```
sparql. setReturnFormat(RDF)
graph = sparql. query(). convert()

per _, _, o nel grafico:
print o
```

sarebbe tornato:

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

Capitolo 5

Fonti di informazione

5.1. Ontologie

Esistono già diverse ontologie di prodotto che possono essere utilizzate per scrivere la propria. Ad esempio:

1. schema.org (https://schema.org/docs/schemas.html) ha una classe per i prodotti e alcune classi più specifiche per articoli specifici, come i libri.
2. Le buone relazioni (http://www.heppnetz.de/ontologies/goodrelations/v1.html) sono specifiche dell'e-commerce e comprendono molti tipi di prodotti.

Altri vocabolari sono disponibili sul sito web Open Linked Vocabularies (https://lov.linkeddata.es/dataset/lov/) all'indirizzo .

5.2. Dati del prodotto

Nel campo della pratica, è difficile ottenere banche dati o servizi ad accesso libero o gratuito con infinformazioni.

Ovviamente, la fonte migliore è amazon.com, che però non offre le informazioni sui prodotti in modo grautomatico ed è molto severo nel raccogliere dati dal proprio sito web tramite bot.

L'uso principale delle sue pagine è quello di definire l'ontologia dei prodotti da utilizzare nella pratica, il sia per la categorizzazione che per i diversi campi utilizzati per descrivere i prodotti. Ovviamente, non è necessario includere tutto quello che c'è.

Una possibilità è quella di generare casualmente un database di prodotti. Una volta definita l'ontologia, non è difficile creare un programma che generi istanze delle diverse classi assegnando valori casuali ai diversi campi all'interno di intervalli che abbiano senso. Nel repository del codice, nella cartella Examples/InfoSources, si trova lo script RandomInfo.py, che può essere utilizzato come esempio.

Capitolo 6

Esempi di agenti

Nella directory AgentExamples del repository sono disponibili tre esempi di agenti di base che apossono aiutare a capire come si implementa un agente e come si effettua la comunicazione.

6.1. Servizio di directory semplice

Questo agente implementa un servizio di directory in cui gli agenti che forniscono servizi possono ressere registrati e ricercati da altri agenti. Questo agente non mantiene la persistenza della directory.

L'implementazione utilizza Flask per consentire la comunicazione via REST. Esistono tre percorsi che implementano il comportamento:

- `/Register`, che consente di registrarsi e cercare nella directory
- `/Info`, che visualizza una pagina web con le richieste di log ricevute (in formato turtle), che possono essere visualizzate ad esempio collegandosi all'agente con un browser.
- `/Stop`, che per l'agente

Gli ultimi due percorsi non richiedono molte spiegazioni. Il primo è quello responsabile della prricezione e dell'elaborazione dei messaggi di registrazione e di ricerca.

Il processo di messaggistica deve seguire il protocollo delle richieste tra agenti. Questo caso è sesemplice, riceviamo un messaggio che non corrisponde a quello che ci aspettiamo, dobbiamo rispondere r dicendo che non l'abbiamo capito, se si tratta di una richiesta di registrazione dobbiamo confermare l'azione e se si tratta di una richiesta di ricerca dobbiamo riferire l'esito.

Il processo che l'implementazione segue è il seguente: il messaggio iniziale viene ricevuto nel parametro content della chiamata a questa voce dell'API, che sarà un messaggio RDF/OWL serializzato in pa XML. Questo messaggio segue il formato FIPA-ACL. Il vocabolario di questa ontologia è importato dalla classe `AgentUtil.ACL` come `ClosedNamespace`, per cui non è possibile verificare che il vocabolario sia usato correttamente. Per poterlo manipolare, deve essere convertito in un grafo RDF. rispetto al parsing del contenuto in una variabile del grafo RDF.

Per facilitare l'elaborazione dei messaggi, la funzione `ACLMessages.get_message_properties` il estrae i campi dal messaggio come un dizionario da cui si possono ottenere i loro valori.

A questo punto dobbiamo decidere quale parte del protocollo eseguire. Se l'estrazione dei campi del messaggio non ha avuto successo, il messaggio non è comprensibile e viene restituito un messaggio di tipo

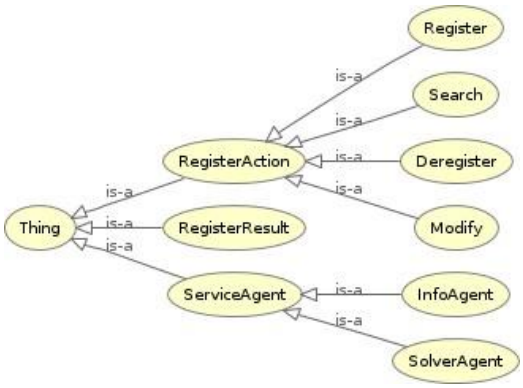

Figura 6.1: Diagramma dell'ontologia del servizio di directory

FIPA-ACL con performativi non compresi. La funzione ACLMessages.build_message è responsabile della costruzione del messaggio FIPA-ACL a partire dai suoi elementi.

Se il messaggio è valido, dobbiamo verificare che abbia il performativo corretto, in questo caso deve del essere un tipo di richiesta. Se non lo è, rispondiamo anche con un non compreso. Se si tratta di una richiesta, dobbiamo
ottenere dal contenuto il tipo di azione richiesta.

Le azioni possibili sono definite nell'ontologia Directory Service Ontology (vedi 6.1), che si trova trar nella directory Ontologies del repository. Questi sono importati dalla classe AgentUtil.DSO, che li definisce come un ClosedNamespace di RDFLib. Ciò consente di verificare che non vengano utilizzati termini dell'ontologia non presenti nel vocabolario. Questo agente implementa solo il processo di due azioni
DSO.Register e DSO.Search.

Se si tratta di un'azione di registrazione, gli elementi necessari per registrare l'agente vengono estratti (la dal messaggio (ddress, Name, Uri, AgentType) e inseriti nella variabile globale dsgraph che contiene un g grafo RDF che memorizza le informazioni di registrazione degli agenti. Queste informazioni sono rappresentate utilizzando DSO e FOAF (ontologia Friend of a Friend), di cui si utilizza la classe FOAF.Agent, che è quella associata all'URI dell'agente e alla relazione FOAF.Name, mentre per il resto dei campi si utilizza DSO. Questi sono i dati minimi che possono essere utilizzati per registrare un agente; il DSO può essere esteso o possono essere utilizzati altri elementi FOAF (o altre ontologie) per aggiungere ulteriori informazioni al registro. Il protocollo di comunicazione in questo caso si conclude con l'invio di un messaggio di conferma performativo per informare l'agente che la registrazione è stata effettuata.

È chiaro che ci sono altri casi che non sono coperti da questo protocollo e che dovrebbero essere a trattati per avere un servizio di registrazione più robusto, come il caso in cui l'agente è già registrato, o quando l'agente ha
qualche errore nei dati dell'agente.

Se si tratta di un'azione di ricerca, l'implementazione considera solo la possibilità di cercare per tipo di ag entità. Questa informazione è uno dei parametri di questa azione nell'ontologia. Possiamo estrarre queste informazioni dal contenuto e interrogare il grafo RDF contenente i record. Viene restituito solo il primo agente trovato, cercando il suo indirizzo nell'URI ottenuto dalla ricerca. Con queste informazioni, viene inviato come risposta un oggetto DSO.Directory-response con l'indirizzo e l'URI dell'agente come parametri. Questa risposta viene inviata come messaggio FIPA-ACL con
informare in modo performativo.

Il resto delle azioni previste dall'ontologia del servizio di directory non sono state implementate, ma il modo per farlo sarebbe simile.

Oltre alle voci API gestite da Flask, si vedrà che viene creato un processo (agentbehavior1) che che consente all'agente di fare altre cose in contemporanea mentre aspetta i messaggi. In questo

In questo caso, questo processo riceve una coda per poter comunicare con altri processi dell'agente. Questa funzione attende solo l'arrivo di informazioni nella coda e termina quando arriva uno zero. Ovviamente mente, possiamo implementare compiti più complessi e avere più compiti concorrenti che fanno cose.

6.2. AgenteInfoSemplice

Questo agente è uno scheletro di agente di informazione/risoluzione che ha anche tre percorsi:

- /comm, che è il comportamento che elabora le richieste in arrivo.
- /iface, che visualizza una pagina web con contenuto fisso.
- /Stop, che per l'agente

All'avvio, l'agente avvia un processo concorrente che registra l'agente con l'agente della directory (presuppone che abbia un indirizzo fisso noto) e attende di leggere dalla coda che riceve come parametro. Questo processo termina quando riceve uno 0 dalla coda. La registrazione dell'agente avviene generando un messaggio FIPA-ACL con richiesta performativa che ha come contenuto un'azione di registrazione dell'ontologia DSO con i dati dell'agente. Per inviare il messaggio, si utilizza la funzione AgentUtil.send_message, che si occupa di serializzare il messaggio, inviarlo all'indirizzo dell'agente di destinazione, attendere la risposta e convertirlo in un grafo RDF.

La voce /comm è in attesa di richieste. Il protocollo di interazione è semplice, i messaggi devono avere la richiesta performativa e devono essere azioni che appartengono all'ontologia che l'agente ha definito (o a un'altra ontologia generale per questo tipo di agenti). L'implementazione risponde un not-understood se il messaggio non è valido o non è una richiesta, altrimenti estrae l'azione dal contenuto e la esegue, restituendo il risultato. Nell'implementazione concreta non viene eseguita alcuna elaborazione dell'azione e risponde sempre con un messaggio performativo inform-done.

6.3. Agente personale semplice

Questo agente è uno scheletro di agente che cerca e utilizza altri agenti per risolvere i problemi; ha tre percorsi

- /comm, che è il comportamento che elabora le richieste in arrivo (non fa nulla).
- /iface, che visualizza una pagina web con contenuto fisso.
- /Stop, che per l'agente

Tutto il comportamento è implementato nel processo concorrente avviato all'inizio dell'agente, ma ovviamente il percorso di comunicazione può essere utilizzato per fare cose più complesse. Questo comportamento esegue una sequenza fissa di azioni che impiega entrambi gli agenti di cui sopra. Per prima cosa cerca l'agente della directory inviando un'azione di tipo DSO.Search per trovare agenti di tipo DSO.HotelAgent (nell'ontologia DSO sono definiti alcuni tipi di agenti). Dal messaggio ricevuto estrae le informazioni necessarie per contattare l'agente che è stato trovato e fa una richiesta che appartiene all'ontologia IAActions (non è definita). Una volta ricevuta la risposta, l'agente termina l'esecuzione di .

6.4. Implementazione degli agenti

Gli agenti di esempio sono implementati in modo da eseguire le azioni descritte sopra. Innanzitutto, deve essere avviato SimpleDirectoryService, che attende le richieste di registrazione. Quindi deve essere avviato SimpleInfoAgent, che si registra nella cartella e attende che

richieste. Infine, deve essere avviato SimplePersonalAgent, che cerca nel servizio di directory e fa una richiesta all'agente informativo. Gli agenti si collegano alle porte 9000, 9001 e 9002 della macchina locale (127.0.0.1). Le voci dell'agente che visualizzano una pagina web accessibile da un browser e la voce che interrompe gli agenti.

Se vogliamo che la comunicazione tra gli agenti avvenga da macchine diverse, dobbiamo cambiare il parametro host del metodo run dell'applicazione Flask in '0.0.0.0'. Da questo momento in poi, il server web accetta connessioni remote. In questo caso, è necessario indicare a ciascun agente l'indirizzo IP (o l'hostname) degli altri agenti con cui deve comunicare.