Documentazione Generatore Tabulati

Margherita Pindaro

October 2021

1 Introduzione

Il generatore è stato realizzato nella fase finale del mio tirocinio, quando il tempo scarseggiava. Per questo motivo ritengo necessarie e suggerisco migliorie al codice, sia in leggibilità, che in testabilità e strutturazione (esempio: creare classe astratta AGENT).

Non mi sono limitata ad applicare la documentazione di Simpy ma ho cercato di mettere su un software riutilizzabile, usando un approccio orientato agli oggetti.

Concetti utili usati da vedere o rivedere: classi con python, design patterns, **map-filter-reduce**, liste python in generale, operatore condizionale (ternaria), regular expression, tuple, `__init__.py`, list comprehension.

Se hai domande: @mpindaro su Telegram.

2 Simulation

Questa è la classe eseguibile. Istanza un oggetto della classe AGENTHANDLER. Richiama i suoi metodi di setup (`agentHandler.create_environment()`) e di inizio della simulazione, specificandone la sua durata:

`agentHandler.start_simulation(env, 15778800)`.

Idealmente in questa classe andrebbero specificati **tutti** i parametri della simulazione, quindi la durata, le frequenze delle chiamate (o di un altro evento), probabilità etc. Attualmente sono *hardwired* nel codice.

3 Agent Handler

La classe più corposa, gestisce tutti gli agenti e la loro interazione. Dato che non avrebbe motivo di esistere più di un agent handler,è un Singleton.

Nell'elenco puntato di seguito verranno spiegato tutti i metodi eccetto i *getter* semplici.

- `get_timestamp_last_state_change(self)`. Restituisce il timestamp dell'ultima volta in cui c'è stato un cambiamento di stato nel sistema.

- `get_timestamp_last_state_change(self)`. Cambia lo stato del sistema con uno nuovo e memorizza il timestamp in cui è avventuo il cambio di stato.
- `register_log(self, timestamp, event)`. Quando si verifica un evento degno di un log, viene richiamato questo metodo, che lo registra insieme all'istante in cui accade. Metodo di utility.
- `create_environment([...])`. Inizializza tutti gli agenti e assegna loro un id.
- `bind(self)`. Associa a ogni agente la lista di altri agenti collegati ad esso. Le relazioni sono simmetriche.
- `__str__(self)`. Overriding del metodo `str`. Tramite essa è possibile vedere tutte le relazioni tra gli agenti. Metodo di utility.
- `start_simulation(self, env, duration)`. Metodo che effettivamente esegue la simulazione. Una volta terminata qui vengono salvati su file in memoria persistente il dataset, i log, e gli agenti coinvolti. Sarebbe opportuno fare queste funzionalità in metodi a se stanti.
- `register_event(self, sender, sender_interc, receiver, receiver_interc,[...])`. Quando si verifica un evento, viene richiamato questo metodo, che registra una riga di dataset. Metodo di utility.
- `generate_sms_cascade(self, sender, sender_interc, receiver, receiver_interc, timestamp)` Nel caso l'evento sia un SMS, tramite questo metodo viene generata una conversazione tra i due agenti. Metodo di utility.
- `handle_call(self, sender, receiver, is_chiamata, duration, timestamp)`. Metodo per gestire una chiamata (o SMS). Nel caso i due agenti siano entrambi non rintracciati l'evento non viene registrato. Se mittente e destinatario sono stessa persona, l'evento non viene registrato. Se l'evento è una chiamata registrerà l'evento altrimenti verrà chiamato `generate_sms_cascade(self, sender,[...])`.

  Infine richiama il metodo che che causa ulteriori eventi in base alle specifiche.
- `innest_events(self, sender, receiver, is_chiamata)`. In seguito a un evento, genera, se necessario, eventi correlati, mandando un interrupt all'agente dovuto.

  Importante: quando viene fatto un interrupt si può specificare una causa (documentazione). In questo caso ho scelto quindi di formattare le stringhe di causa nel seguente modo: <TIPOLOGIA_EVENTO>-<CLASSE_AGENTE><ID_AGENTE_MITTENTE>. Quindi, consideriamo un caso in cui un importatore (sender), con id 23, chiama un camionista (receiver). Il camionista in seguito a una chiamata di un importatore chiama uno spacciatore. In questo caso quindi la stringa della causa sarà *chiamata-importatore23*.

Questo metodo contiene un errore sulle condizioni, ho messo tutti receiver ma alcune dovrebbero coinvolgere il sender. Da correggere in base alle specifiche.

- `get_call_param(self, list_of_receivers)`. Per comodità i parametri delle chiamate (mittente, destinatario, durata etc.) sono creati in questa classe, per un qualsiasi agente. La lista dei destinatari è solitamente una lista di liste. Metodo di utility.

4 States

Enumerativo. Contiene gli stati del sistema.

5 Agenti

Come detto prima, è necessario la creazione di una classe astratta, *parent* di tutte le classi di agenti. Non spiegherò tutti i metodi di tutte le otto classi, ma solo quelli che trovo significativi. Prenderò d'esempio la classe IMPORTATORE, ma i commenti sono generali e valgono per tutte.

- `__str__(self)`. Mostra tutti gli agenti con il quale l'agente è collegato.
- `enter_simulation_environment(self, importatori, esportatori, spacciatori. [...])`. Setup dell'agente, acquisce le sue relazioni e i suoi parametri.
- `__eq__(self, o: object) -> bool`. Consideriamo due agenti uguali se hanno lo stesso id.
- `doIKnowPersonX(self, id)`. Metodo che restituisce l'id dell'agente se esso ha, nei propri vettori di relazione, l'agente con id specificato via parametro.
- `call_someone(self, is_chiamata, duration, receiver)`. Metodo semplice che richiama il metodo `handle_call` della classe AgentHandler.
- `change_cella(self)`. Aggiorna la posizione corrente dell'agente.
- `run(self)`. Documentazione Simpy