Laboratorio di Programmazione

Stefano Russo Andrea Mecchina Michele Rispoli
Pietro Morichetti Nicolas Solomita

11 Dicembre 2020 - Foglio 2 - versione 1.1

1 Eccezioni e Sanitizzazione

Esercizio 01

Il centro di ricerca sullo sviluppo di intelligenze artificiali di Dublino vi ha assegnato il compito di sviluppare uno script in Python (aka un programma) per un automa: l'androide deve simulare il comportamento umano dell'atto di vestirsi. La normale procedura che tutti noi eseguiamo in maniera automatica può essere riassunta nei seguenti steps (funzioni):

1. *biancheria*, atto di indossare la biancheria intima; la funzione restituisce 1 se ha avuto successo, altrimenti 0.
2. *calzini*, atto di indossare i calzini; la funzione restituisce 1 se ha avuto successo, altrimenti 0.
3. *maglia*, atto di indossare una maglietta; la funzione restituisce 1 se ha avuto successo, altrimenti 0.
4. *pantaloni*, atto di indossare i pantaloni;la funzione restituisce 1 se ha avuto successo, altrimenti 0.
5. *calzatura*, atto di indossare le scarpe; la funzione restituisce 1 se ha avuto successo, altrimenti 0.

Sviluppare una classe **Automa** che presenta come attributi i capi di vestiario sopra citati, tutti inizializzati a None ed un metodo per ogni azione richiesta dall'automa. Si osservi che in caso di successo per un'azione, il corrispondente attributo è impostato a True (Es. l'automa prova ad indossare la maglietta, l'azione ha avuto successo e prima che il metodo ritorni 1, l'attributo self.maglia diventa True).

Il corpo principale dello script non dovrà vestire l'automa in maniera procedurale ma piuttosto casuale sfruttando un while; di seguito vi viene proposto una porzione di pseudo-codice, non esaustiva:

```
import random

class Automa{
    ...
}

def esegui(automa, capo){ # funzione esterna
    # in base al valore dell'ingresso
    # richiama il metodo della classe associata
    # al capo di abbigliamento

    # ritorna quanto restituito dal metodo chiamato
}

...
capi_vestiario = [...]
vestito = True
...

while(vestito){
    # seleziona a caso un capo
    # (G1) gestire in caso il capo selezionato
    # va contro la sequenza della procedura
    # chiama esegui per provare ad indossare il capo
    # (G2) gestire in caso di insuccesso
}

print('automa vestito correttamente')
```

Il concetto di *casualità* all'interno di un linguaggio di programmazione è espresso per mezzo di funzioni apposite, che sono state fornite in calce all'esercizio.

Osserviamo che esistono due situazioni in cui si potrebbe avere un comportamento scorretto da parte dell'androide:

- **G1** - il capo selezionato (casualmente) non può essere indossato poiché non rispetta la naturale procedura (es. indossare pantaloni prima della biancheria). Questo tipo di anomalia può essere gestito attraverso il costrutto *if*: non sanitizza ma si limita a riportare l'errore ed il capo non viene indossato.
- **G2** l'automa ha tentato di indossare il capo ma non ci è riuscito (il metodo ha restituito 0 in maniera casuale). Questo tipo di errore è per

noi un errore "fatale" che non possiamo gestire; si utilizzi il costrutto *raise* riportando l'errore.

Note Importare la libreria **random** per poter utilizzare le seguenti funzioni di casualità:

- *random.choice(lista)*, la funzione estrae casualmente e restituisce un elemento della lista passata come argomento (la lista resta immutata).
- *random.randint(a, b)*, la funzione restituisce un intero compreso nell'intervallo $[a, b]$.
- altre funzioni nel seguente link funzioni random in Python.

Siete invitati a gestire anche casi di errore provenienti da queste funzioni.

Esercizio 02

Volete implementare un' IA che giochi a scacchi. I dati forniti in input all'algoritmo saranno delle sequenze di mosse effettuate da entrambi i giocatori, e durante le partite l'algoritmo dovrà pur comunicarci cosa intende fare, dunque dovremo scegliere una rappresentazione per le mosse. Fortunatamente esiste già una codifica universalmente riconosciuta (Notazione Algebrica su Wikipedia) che per semplicità scegliamo di adottare. Ogni posizione nella scacchiera corrisponde a una coppia di coordinate ($\{a, \ldots, h\}$ per le ascisse, $\{1, \ldots, 8\}$ per le ordinate) e ad ogni mossa è associata una stringa, in cui sono indicati:

1. Il pezzo che si sta muovendo, riportandone l'iniziale in maiuscolo (`R`e, `D`onna, `T`orre, `A`lfiere e `C`avallo, per il pedone non si mette niente)
2. Eventualmente si indica una delle coordinate della posizione di partenza, se necessario disambiguare
3. Se è stata effettuata una cattura si indica con una `x`
4. La posizione di arrivo del pezzo, in coordinate alfanumeriche
5. Eventuali ulteriori annotazioni per indicare la cattura en passant dei pedoni (si nota `e.p.`), la promozione dei pedoni (indicando la lettera del pezzo risultante) o per lo scacco (con un `+` terminale).

Nel caso in cui la mossa sia un *arrocco* lo si indica con una notazione speciale:

- `0-0` per l'arrocco breve
- `0-0-0` per l'arrocco lungo.

Ecco alcuni esempi:

- `Dxg4` : Donna cattura in g4
- `e2` : Pedone muove in e2
- `xf5` : Pedone cattura in f5
- `Tb3` : Torre muove in b3

Scrivere in Python una funzione `checkChessSyntax( text )` che prenda input una **stringa** e controlli che sia una mossa sintatticamente valida secondo la notazione appena introdotta. Se l'espressione è valida la funzione non deve ritornare nulla. Vogliamo invece che la funzione lanci un'eccezione nel caso in cui uno dei seguenti controlli fallisse:

- L'input deve essere una stringa
- La stringa è ben fomata, ovvero ricade in uno dei seguenti casi:
  1. È esattamente "`0-0`" o "`0-0-0`"
  2. Per semplicità ci limiteremo a considerare espressioni della forma "[*Pezzo*][*flag_cattura*] *coordinate_arrivo*", dove
     - *Pezzo* è un carattere tra `R`, `D`, `T`, `A`, `C` oppure è assente (nel caso del pedone)
     - *flag_cattura* è `x` oppure è assente
     - *coordinate_arrivo* è una coppia di coordinate alfanumeriche.

Il controllo sulle stringhe per l'arrocco può essere fatto comparando direttamente la stringa con le espressioni letterali "`0-0`" o "`0-0-0`" (i.e. `if text=='0-0' ...`). In caso non vi sia un match possiamo controllare la validità dell'espressione, senza impazzire con `for` e `if` vari, sfruttando le Regular Expression, strumento potentissimo ed onnipresente per risolvere problemi di pattern matching, per cui esiste un modulo specifico all'interno di Python.

Le RegEx ci permettono di trovare le occorrenze di una stringa o di un pattern (come quello del nostro problema) all'interno di un altra stringa. In questo caso basterà controllare che la stringa soddisfi la seguente espressione regolare:

```
^[RDTAC]?x?[a-h][1-8]$
```

Qui potete testare il funzionamento di questa espressione regolare interattivamente (vi consiglio di tenere questo sito nei bookmark, torna parecchio utile), qui un crash course sulle RegEx per chi volesse un'introduzione pratica, ed infine qui un piccolo tutorial sul modulo Python `re` per chi volesse approfondire come utilizzarle nel codice.

Per effettuare il controllo all'interno del nostro codice sarà sufficiente importare il modulo `re` e utilizzare la funzione `re.match(pattern,input)`, la quale restituirà `None` nel caso in cui NON vi sia il match, come nell'esempio:

```
import re # importo il modulo per usare le RegEx

pattern = '^[RDTAC]?x?[a-h][1-8]\$' # salvo il pattern in una variabile

print(re.match(pattern,'Dxf3') is not None)
    #> True, l'espressione è valida

print(re.match(pattern,'mossapotentissima') is not None)
    #> False, non è un'espressione valida
```

Qui trovate il template già pronto con annesso test per controllare la validità della vostra soluzione. Sentitevi liberi di forkare il repo (o di scriverci per chiarimenti se quest'ultima frase vi sembrasse arabo).

2 Testing

Esercizio 03

Definire una classe **Calcolatrice** che realizza le principali funzioni del calcolo aritmetico (somma, sottrazione, moltiplicazione e divisione) e anche le funzioni di potenza, di modulo, di radice e conversione di base. In particolare si osservi che:

- non sono ammesse operazioni tra oggetti che che non siano parte delle note classi numeriche (Integer, Float, ...); a meno che i valori numerici siano passati in forma di stringa;
- divisione per zero non è ammesso;
- potenza ammessa solo con base e potenza interi;
- l'operazione di potenza ammette base negativa;
- l'operazione di radice è ammessa solo per valori strettamente positivi;
- l'operazione di conversione di base è esclusivamente dalla base 10 a base 2;
- sanitizzare dove possibile.

Oltre alla classe precedente, realizzare una classe **Test** che andrà a testare tutti i metodi della classe Calcolatrice; in caso di errori, la classe deve stampare su schermo un messaggio di avvertimento con tutte le informazioni necessarie.

Provare a completare l'esercizio seguendo la filosofia del *test-driven*.

Esercizio 04

Nella lezione 6 abbiamo modificato l'oggetto *CSVFile* in modo che la funzione **get_data(self, start=None, end=None)** permetta di leggere un particolare intervallo di righe del file. Nel caso in cui *non* vengano passati i parametri *start* e *end*, vogliamo che la funzione ritorni *tutte* le righe del file.

Scrivere una funzione di test che verifichi che il comportamento del codice modificato sia quello desiderato. Per esempio, un'idea potrebbe essere creare una funzione **test_sum(file)** che confronta:

- la lista contente i valori di tutte le righe ottenuta tramite **file.get_data()**,
- la lista contenente i valori di tutte le righe ottenuta fornendo esplicitamente gli estremi delle righe, tramite **file.get_data(1,size)** dove *size* è il numero di righe totali contenute nel file.

Un possibile criterio di confronto potrebbe essere verificare che la somma dei valori di entrambe le liste coincida. A questo propositivo è possibile riciclare la funzione **list_sum(the_list)** vista durante la lezione 3, la quale ritorna proprio la somma degli elementi della lista *the_list*. Nel caso in cui il criterio di confronto non sia rispettato, la funzione di test deve lanciare un'eccezione.

Un ulteriore possibile controllo potrebbe essere verificare che la somma degli elementi della lista che seleziona solo un numero valido di righe sia minore o uguale alla somma di tutti gli elementi della lista.