

Programmazione: Modulo 1 – Laboratorio Di Programmazione

A.A. 2022/23

Relazione

Nella realizzazione del progetto si è scelto di seguire due fasi distinte: nella prima, è stata implementata la logica del gioco, utilizzando la riga di comando per permettere le scelte e gli input dell'utente; nella seconda, è stata realizzata una semplice interfaccia grafica modificando alcune delle funzioni del codice già scritto.

FASE 1: Implementazione della logica del gioco della vita.

La prima scelta progettuale riguarda la definizione di due costanti per identificare le celle vive e le celle morte, in modo che sia sempre possibile cambiare il simbolo utilizzato per rappresentarle, senza modificare tutto il codice.

Innanzitutto, il programma chiede all'utente se iniziare caricando una precedente simulazione (scelta 1) o se creare una nuova simulazione, quindi una nuova griglia (scelta 2). Nel primo caso viene chiamata la funzione `carica_simulazione_da_file()`, che imposta un loop nel quale:

- si chiede all'utente di inserire il nome del file;
- viene tentata l'apertura del file con il nome fornito; se l'operazione (`f = open(nomefile)`) avviene con successo, il contenuto del file viene letto e memorizzato nella variabile `griglia` (`griglia=f.read()`). La griglia è rappresentata come una lista di liste, dove ogni lista interna rappresenta una riga della griglia.
- viene assicurato che il formato della griglia sia omogeneo (i caratteri diversi da `cella_viva` vengono convertiti in `cella_morta`). In questo modo ho disaccoppiato il file caricato come simulazione dalla logica, ovvero dal sistema adottato per distinguere le celle morte, e non devo riscriverlo ogni volta: in questo modo al posto di X e O posso utilizzare qualsiasi simbolo o carattere.
- viene fatto un `return` della griglia ottenuta.

Nel secondo caso (scelta 2), viene chiamata la funzione `genera_griglia()`, che provvede alla generazione di una nuova griglia in base alla lunghezza del lato e alla percentuale di riempimento specificate dall'utente tramite le funzioni `carica_lato_griglia()` e `carica_percentuale_riempimento()`. Nella prima verifico due condizioni, ovvero che il valore inserito sia una stringa formata solo da cifre numeriche (`isdigit`) e che sia maggiore di 0; nella seconda, invece, il valore inserito dall'utente deve essere un numero reale compreso tra 0 e 1. Con questi parametri, viene inizializzata una griglia vuota (`griglia=list(' '*(l*l))`), fatta di singoli spazi vuoti e lunga quanto il numero complessivo di "caselle" della griglia (`l*l`). Viene poi calcolato il numero di celle vive in base al coefficiente di riempimento e viene memorizzato nella variabile `num_vive` (`num_vive=round(sigma*l*l)`). Per posizionare casualmente le celle vive all'interno della griglia, è stato utilizzato il modulo **Random** e creato un ciclo che, per un numero equivalente al numero di celle vive, sistema casualmente nella lista una cella viva: ogni volta, viene assegnata a una cella viva una posizione (variabile `p`) casuale scelta tra tutte quelle posizioni che non sono già occupate da celle vive. In sostanza, la variabile `p` corrisponde a una posizione casuale tra 0 e il numero massimo possibile di celle della lista; viene quindi assegnata la posizione a una cella viva verificando che non sia già stata assegnata. Si ottiene così una nuova stringa `griglia2` di singoli caratteri, che sono suddivisi in righe inserendo `\n` nei punti opportuni,

grazie a un secondo ciclo; con un ultimo ciclo, infine, ciascuna riga viene poi convertita in una lista di caratteri (`griglia2[i]=list(griglia2[i])`). In questo modo, alla fine di questo blocco di codice, `griglia2` è una lista di liste (così come `griglia`), e rappresenta la griglia generata dall'utente. La funzione restituisce la griglia generata (`return griglia2`).

Poiché il numero di passi deve essere definito dall'utente in entrambi i casi, sia con una griglia nuova (scelta 2) sia con una simulazione preesistente (scelta 1), dopo la scelta iniziale chiamo una funzione (`carico_numero_passi()`) per sapere il numero di passi stabiliti dall'utente. Anche in questo caso, dal momento che l'input deve essere un numero intero maggiore di 0, stabilisco una condizione (`if not numero_passi.isdigit() or not int(numero_passi) > 0:`) che se verificata produce un messaggio di errore.

Una volta che è stata definita la griglia di partenza, con la sua prima generazione, il gioco può avere inizio. Da qui si chiamano le funzioni `disegna_griglia()` e `animazione()` che permettono di visualizzare graficamente la griglia e passare da una generazione all'altra (si veda §[FASE 2: Implementazione della GUI](#)).

Per passare da una generazione all'altra sono state implementate due funzioni principali:

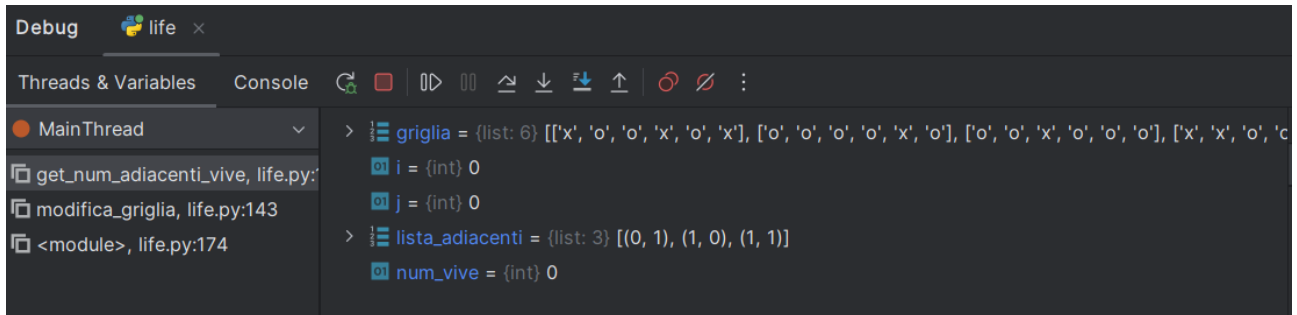
1. `get_num_adiacenti_vive(griglia, i, j)`
2. `modifica_griglia(griglia)`

Condizione imprescindibile per effettuare il passaggio generazionale è quella, infatti, di accertarsi preventivamente del numero di celle vive adiacenti a ciascuna cella: bisogna quindi scorrere tutta la griglia, cella per cella, e verificare la condizione delle 8 celle adiacenti. Questo è il compito della funzione (1) `get_num_adiacenti_vive(griglia, i, j)`. Per ogni cella, che ha posizione identificata da due indici (`i, j`), è quindi necessario registrare il numero complessivo di celle vive o morte attorno ad essa. La funzione è progettata in modo che passando come parametri la griglia e la posizione della cella (`i, j`), essa restituisca il numero di cellule vive (`num_vive`). Preventivamente, bisogna calcolare la lista delle celle adiacenti (`lista_adiacenti`), in cui ogni elemento contiene le coordinate di una cella adiacente a una data cella nella griglia; questa lista è necessaria per via della presenza di tutte quelle celle che si trovano sul bordo della griglia. Se la cella si trova sui bordi della griglia, infatti, non ha 8 celle adiacenti, ma un numero variabile caso per caso:

- quando la cella non si trova nella prima riga [`i>0`] vanno considerate tutte le celle della riga superiore [`lista_adiacenti.append((i-1, j))`]; se non si trova nella prima colonna [`j>0`], si considerano anche la cella in alto a destra [`lista_adiacenti.append((i-1, j-1))`] e quella sulla stessa riga a destra [`lista_adiacenti.append((i, j-1))`]; se non si trova nell'ultima colonna [`j<len(griglia[0])-1`], si considerano la cella in alto a sinistra [`lista_adiacenti.append((i-1, j+1))`] e quella sulla stessa riga a sinistra [`lista_adiacenti.append((i, j+1))`].
- Quando la cella si trova nella prima riga (`else:`), si considerano solo le celle della stessa riga, a sinistra [`lista_adiacenti.append((i, j-1))`], se la colonna non è la prima) o a destra [`lista_adiacenti.append((i, j+1))`], se la colonna non è l'ultima).
- Quando la cella non si trova nell'ultima riga [`if i<len(griglia)-1:`] vanno considerate tutte le celle della riga inferiore [`lista_adiacenti.append((i+1, j))`]; se la cella non si trova nella prima colonna, inoltre, si considera anche la cella in basso a sinistra

```
[lista_adiacenti.append((i+1, j-1))]; se non si trova nell'ultima colonna, si  
considera anche la cella in basso a destra [lista_adiacenti.append((i+1, j+1))].
```

Si può verificare che le condizioni siano quelle giuste e sufficienti a raccogliere tutte le celle adiacenti tramite debug; ad esempio:

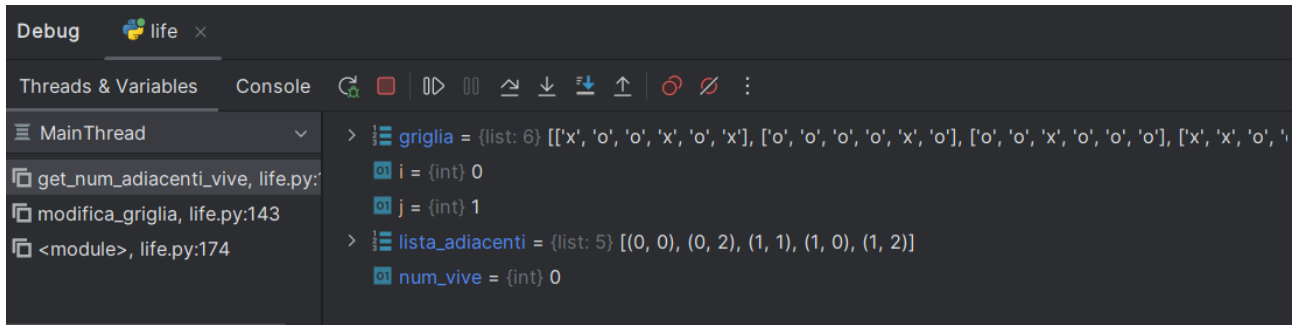


```
Debug life x
Threads & Variables Console
MainThread
get_num_adiacenti_vive, life.py:
modifica_griglia, life.py:143
<module>, life.py:174
> griglia = (list: 6) [['x', 'o', 'o', 'x', 'o', 'x'], ['o', 'o', 'o', 'o', 'x', 'o'], ['o', 'o', 'x', 'o', 'o', 'o'], ['x', 'x', 'o', 'c
i = (int) 0
j = (int) 0
> lista_adiacenti = (list: 3) [(0, 1), (1, 0), (1, 1)]
num_vive = (int) 0
```

La schermata di debug mostra come per la cella in posizione (0,0) – cioè l'angolo in alto a sinistra – le celle adiacenti siano solo 3:

- quella in posizione (0,1): ovvero stessa riga, colonna accanto, a sinistra
- quella in posizione (1,0): ovvero riga sotto, stessa colonna, la prima
- quella in posizione (1,1): ovvero riga sotto, seconda colonna.

Per la cella in posizione (0,1), ovvero la seconda cella della prima riga, le adiacenti sono 5 [(0,0), (0,2), (1,1), (1,0), (1,2)] perché non esistono le tre celle della riga superiore:



```
Debug life x
Threads & Variables Console
MainThread
get_num_adiacenti_vive, life.py:
modifica_griglia, life.py:143
<module>, life.py:174
> griglia = (list: 6) [['x', 'o', 'o', 'x', 'o', 'x'], ['o', 'o', 'o', 'o', 'x', 'o'], ['o', 'o', 'x', 'o', 'o', 'o'], ['x', 'x', 'o', 'c
i = (int) 0
j = (int) 1
> lista_adiacenti = (list: 5) [(0, 0), (0, 2), (1, 1), (1, 0), (1, 2)]
num_vive = (int) 0
```

Queste condizioni permettono di creare una lista di coordinate che identificano tutte le celle adiacenti per ciascuna cella.

Attraverso il ciclo `for adiacente in lista_adiacenti`: si scorre ogni elemento nella lista `lista_adiacenti`; per ogni elemento vengono estratte le coordinate `adiacente_i` e `adiacente_j` che indicano la posizione di una cella adiacente nella griglia rispetto alla cella di partenza; e attraverso le nuove coordinate si verifica se la cella adiacente nella griglia è viva: in caso positivo, si incrementa di uno la variabile `num_vive`.

Nella funzione (2) `modifica_griglia(griglia)` va creata una nuova variabile che possa registrare il nuovo stato della griglia (senza sovrascrivere quello precedente), dal nome `griglia_successiva`, inizializzata come lista di liste, la stessa tipologia di dato della variabile `griglia`.

Per scorrere la griglia “originale” (quella cioè caricata o creata dall’utente) – o meglio, su quella corrente, perché poi questa funzione verrà chiamata anche su altre griglie, per ogni passaggio da una generazione all’altra – è stato implementato un ciclo che scorre righe e colonne,

```
for i in range(len(griglia)):
    for j in range(len(griglia[0])):
```

In questo ciclo, si stabiliscono diverse condizioni a seconda che la cella sia viva (`if griglia[i][j]==cella_viva:`) oppure morta (`else:`); in ciascuno dei due scenari, poi, lo stato della cella viene modificato in base alle condizioni fornite nella descrizione del *gioco della vita*:

1. Qualsiasi cella viva con meno di due celle vive adiacenti muore, come per effetto d’isolamento → questa condizione non viene gestita perché le nuove celle sono già inizializzate su “morta”
2. Qualsiasi cella viva con due o tre celle vive adiacenti (`if num_adiacenti_vive in (2,3):`) sopravvive alla generazione successiva → quindi da morta, la cella diventa viva (`nuova_riga[j]=cella_viva`)
3. Qualsiasi cella viva con più di tre celle vive adiacenti muore, come per effetto di sovrappopolazione → questa condizione non viene gestita perché le nuove celle sono già inizializzate su “morta”
4. Qualsiasi cella morta con esattamente tre celle vive adiacenti (`if num_adiacenti_vive==3:`) diventa una cella viva (`nuova_riga[j]=cella_viva`), come per effetto di riproduzione

Create le nuove righe, queste vengono poi aggiunte alla lista `griglia_successiva` tramite la funzione `.append()`.

Infine, è stato creato un ciclo `for` che ripete la funzione `modifica_griglia` per tante volte quanti sono i passi stabiliti dall’utente, sovrascrivendo ogni volta l’ultima versione della griglia a quella precedente, in modo che le modifiche tra una generazione e l’altra avvengano sempre sulla griglia aggiornata.

Per salvare la simulazione su file, è stata implementata la funzione `salva_su_file`, che prende come parametri in ingresso `griglia` e `nomefile`. Questa funzione si occupa di salvare l’ultima configurazione della griglia in un file specificato. Nello specifico, viene passata in un ciclo `for` ogni riga della griglia, che viene convertita in stringa e concatenata alle altre insieme a un’interruzione di riga. La funzione `salva_su_file` è utilizzata all’interno del codice di gestione di un evento dell’interfaccia grafica.

FASE 2: Implementazione della GUI.

Modulo Tkinter

L’interfaccia per la parte iniziale del programma (ovvero il caricamento della simulazione, l’inserimento dei parametri per la creazione di una nuova griglia ecc.) e per la parte finale (eventuale salvataggio su file) è stata creata con il modulo **Tkinter** (libreria standard disponibile su Python).

Sono state utilizzate tre diverse classi di widget:

- **Label**: finestra con testo
- **Button**: pulsante contenente testo, che reagisce al clic del mouse
- **Entry**: una casella di testo per l’input dei parametri

Due diverse tipologie di gestore di eventi (event handler) sono state utilizzate:

- **Clic del mouse**: per il widget `button`
- **Input di testo**: per il widget `entry`

Modulo Turtle

È stato utilizzato il modulo **Turtle** per visualizzare graficamente la griglia.

La griglia è composta da una serie di quadrati affiancati che corrispondono a una cella: se la cella è morta il quadrato è vuoto (il puntatore disegna solo i bordi del quadrato), se la cella è viva il quadrato è pieno.

Per disegnare il singolo quadrato sono stati utilizzati i comandi:

1. `turtle.pendown()` → abbassa la penna della tartaruga, indicando che la tartaruga dovrebbe iniziare a disegnare quando si muove. Segue un ciclo (`for _ in range(4):`) che si ripete quattro volte. L'underscore è utilizzato perché non è usato alcun valore di iterazione del ciclo (l'underscore come convenzione in Python per indicare una variabile che non viene utilizzata).
2. `turtle.forward(square_width)` → fa avanzare la tartaruga in avanti di una distanza pari a `square_width`. In questo caso, `square_width` rappresenta la lunghezza di ciascun lato del quadrato ed è stata definita precedentemente (con valore 20, ad esempio).
3. `turtle.right(90)` → fa ruotare la tartaruga di 90 gradi in senso orario. Alla fine del ciclo `for`, avrò disegnato un quadrato completo e la tartaruga sarà posizionata nella stessa posizione in cui ha iniziato a disegnare il quadrato.
4. `turtle.penup()` → Questo comando alza la penna della tartaruga, indicando che la tartaruga può muoversi senza disegnare.

Per disegnare la griglia sono stati allineati tanti quadrati quanti dichiarati dalla variabile `l` (lato), sia in larghezza che in lunghezza, con i seguenti comandi:

1. `turtle.forward(spacing)` → Questo comando sposta la tartaruga avanti di una lunghezza pari alla variabile `spacing`. Per fare in modo che il disegno abbia l'aspetto di una griglia, i quadrati sono stati affiancati, riducendo il valore della variabile `spacing` a un valore equivalente a quello di `square_width` (20).
2. `turtle.backward(spacing * n)` → Questo comando riporta la tartaruga al punto di partenza
3. `turtle.right(90)` e `turtle.forward(spacing)` → Comandi per girare la tartaruga verso il basso e spostarla di una lunghezza pari a quella del lato del quadrato; poi la tartaruga viene girata di nuovo nella posizione giusta per cominciare una nuova riga con `turtle.left(90)`

Viene creato un ciclo (`for row in range(n):`) che permette di creare una griglia di lato `n`.

Per disegnare le celle vive è stato utilizzato il metodo `fill` del modulo Turtle: il modo in cui viene costruito il quadrato è lo stesso, ma viene introdotto dal comando `turtle.begin_fill()` e concluso dal comando `turtle.end_fill()`. Per praticità, è stata creata una funzione a parte `filled_square(width)`: che viene chiamata al bisogno.

Le funzioni base di Turtle sono state utilizzate all'interno delle funzioni:

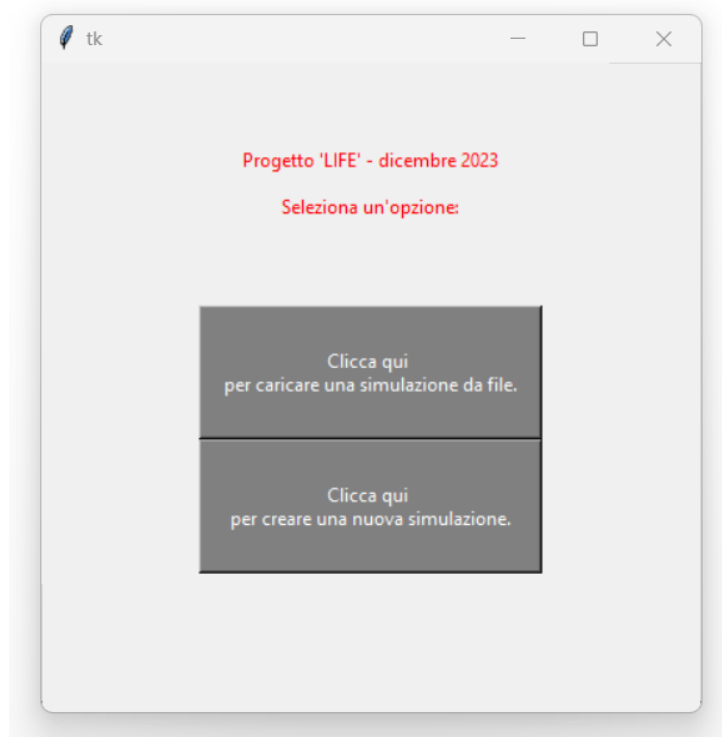
1. `disegna_griglia()` → funzione che disegna la griglia sulla finestra grafica. A partire dal parametro `griglia`, è stato costruito una doppia iterazione `for` che scorre ogni cella della griglia. Se la cella è viva (`cella_viva`), chiama la funzione ausiliaria `filled_square()` per disegnare

un quadrato pieno di dimensioni specificate da `square_width`; se la cella è morta (`cella_morta`), disegna un quadrato vuoto usando le funzioni base del modulo Turtle.

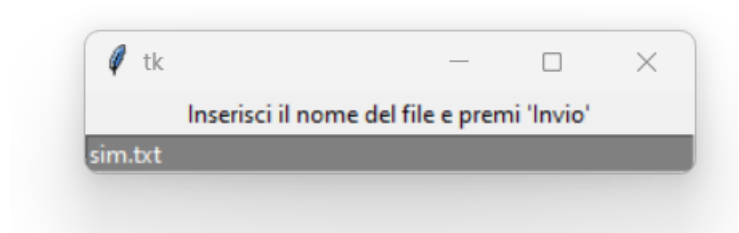
2. `animazione()` → funzione che, attraverso un ciclo, gestisce il passaggio da una generazione di cellule all'altra. Vengono passati i parametri `griglia` e `numero_passi`, e in base a questi viene chiamata la funzione `modifica_griglia` per definire la nuova generazione della griglia in base alle regole fornite dalla descrizione del *gioco della vita* (la funzione è descritta nel dettaglio in § [FASE 1: Implementazione della logica del gioco della vita](#)). Viene quindi chiesto di mostrare la nuova generazione tramite la funzione `disegna_griglia()` e poi si sovrascrive la griglia corrente per poter poter reiterare i passaggi per un numero di volte pari al valore di `numero_passi`.

Manuale utente

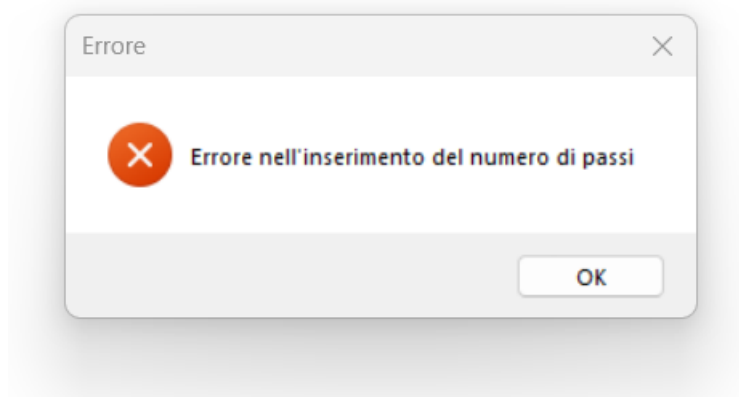
L'interfaccia utente viene attivata al comando "Run" di un ambiente di sviluppo quale PyCharm. L'interfaccia apre una finestra che può essere ridimensionata a piacere.



Per interagire con l'interfaccia, occorre utilizzare il mouse e cliccare sui pulsanti per effettuare le scelte. L'interfaccia prevede anche l'inserimento di testo da tastiera, in una casella di testo che si attiva al clic del mouse sull'apposito riquadro.



Se si sceglie di caricare la simulazione che è stata appositamente creata, come da indicazioni, dovrà essere inserito il testo “sim.txt”. Se invece si opta per la creazione di una nuova griglia, verrà chiesto all’utente di inserire la lunghezza del lato della griglia e il coefficiente di riempimento. Se viene inserito un valore sbagliato, viene generato un messaggio di errore: l’applicazione continua quando si clicca su “ok”, dando la possibilità all’utente di inserire nuovamente il valore nella casella di testo.



La griglia con le celle e i passaggi da una generazione e alla successiva vengono quindi generati autonomamente, senza che la necessità di altri input da parte dell’utente (con un intervallo di un secondo tra una generazione e l’altra). Al termine del processo, l’applicazione si ferma sull’ultima generazione, e chiede all’utente se desidera procedere con il salvataggio su un file in formato txt. L’applicazione chiude infine tutte le finestre.