

Appunti di scuola

PygameZero

Diomede Mazzone

2021, ver. 0.2



Sommario

Sommario	1
Introduzione	2
Prerequisiti e primo utilizzo	2
Installazione	2
Prendi l'alieno	3
Actor	4
Click del mouse	7
Ordine nello script	8
Messaggi in sovrapposizione	9
Esercitazione	10
Mangia le monete	11
Variabili	12
Actor	12
Gestione del tempo	14
Variabili locali e variabili globali	14
Game over	15
update()	15
Input da tastiera	16
Esercitazione	16
Unisci le stelle	18
Variabili di tipo lista	19
Ciclo di for	19
Disegnare una linea	21
Tuple	22
Gestione del tempo con time()	23
Fase di test	25
Esercitazione	26

Introduzione

Il presente documento si pone come obiettivo la realizzazione di semplici giochi attraverso l'uso della libreria Pygame Zero, in Python 3. Gli script sono a titolo di esempio, per maggiori dettagli si faccia riferimento alla documentazione ufficiale:

<https://pygame-zero.readthedocs.io/en/stable/index.html>.

I dettagli della sintassi relativa a python e tutti i costrutti devono essere approfonditi in altra sede.

Prerequisiti e primo utilizzo

Per svolgere in modo consapevole gli esercizi è sufficiente la conoscenza dei costrutti basilari della programmazione in Python: selezione, cicli, uso delle variabili ed indentazione. Tali concetti verranno talvolta ripresi, anche se in modo sintetico.

Installazione

Pygame Zero è un modulo aggiuntivo di Python, quindi sul proprio pc dovrà essere presente Python 3.x ed il suo *idle*. Per installare moduli aggiuntivi è sufficiente utilizzare **pip**, il gestore di pacchetti di Python interno al sistema stesso. Se non si dovesse essere certi di averlo installato, è possibile verificare l'esistenza ed eventualmente installarlo. Si apra il prompt dei comandi in Windows (oppure il terminale su sistemi Mac o Linux) e digitare:

A screenshot of a terminal window. The title bar shows 'diomedede@diomedede:~'. The prompt is '(base) [diomedede@diomedede ~]\$' and the command entered is 'python -m pip install -U pip'.

```
diomedede@diomedede:~  
(base) [diomedede@diomedede ~]$ python -m pip install -U pip
```

Attraverso pip si deve prima installare **Pygame**, successivamente **Pygame Zero**, versione semplificata del primo modulo.



```
diomedede@diomedede:~  
(base) [diomedede@diomedede ~]$ pip install pygame  
  
diomedede@diomedede:~  
(base) [diomedede@diomedede ~]$ pip install pgzero
```

Per verificare l'installazione corretta di tutti i pacchetti è sufficiente costruire un primo piccolo ambiente di gioco, così da verificarne il funzionamento. Sarà sufficiente quindi creare una cartella sul desktop all'interno della quale salvare un file di nome **test.py**.

La prima funzione da creare è **draw()** che permette al sistema di comprendere cosa deve disegnare, creare l'ambiente di gioco ed eventualmente disegnare altri elementi al suo interno.

```
4  
5 def draw():  
6     screen.clear()  
7 |
```

Per eseguire lo script è necessario portarsi con il terminale all'interno della cartella sul desktop e digitare: **pgzrun test.py**. Sarà visualizzata una schermata nera, sarà il campo di gioco.

```
(base) [diomedede@diomedede pgZero]$ pgzrun prendiAlieno.py  
pygame 2.0.1 (SDL 2.0.14, Python 3.8.8)  
Hello from the pygame community. https://www.pygame.org/contribute.html  
█
```

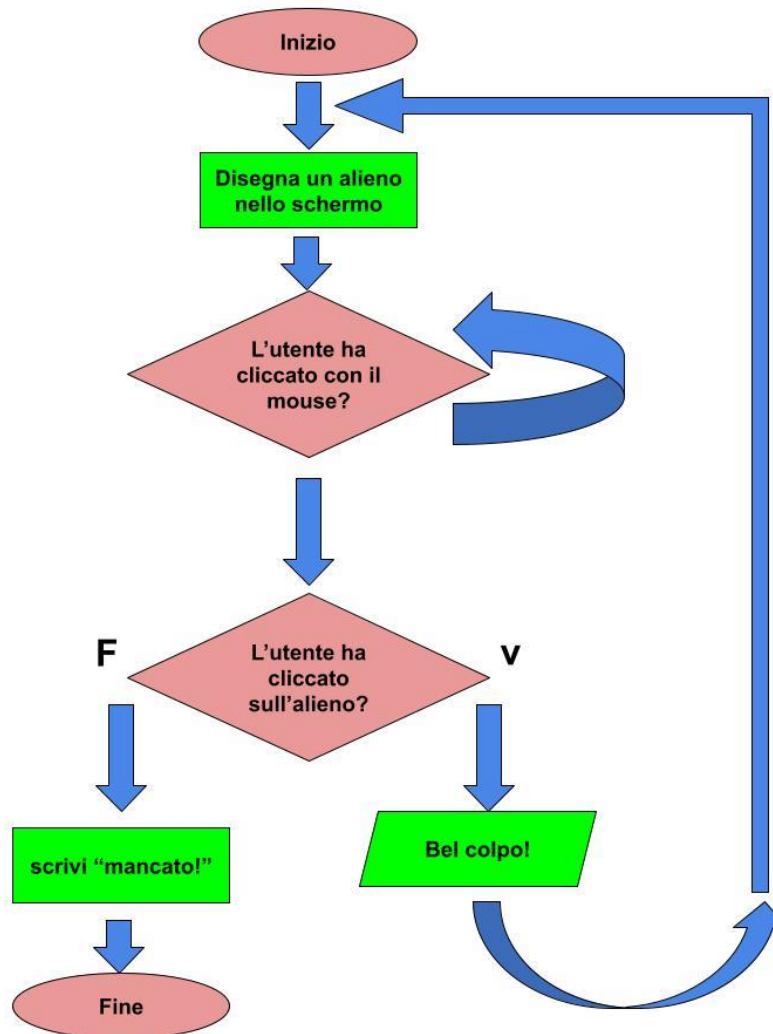


Prendi l'alieno

Pygame zero ragiona per eventi, gli script vengono eseguiti in un ciclo infinito e soltanto un evento, deciso dal programmatore, può arrestare il ciclo ponendo fine al gioco.

Innanzitutto bisogna immaginare il funzionamento del gioco da realizzare ed è buona norma costruire un diagramma di flusso che definisce gli eventi possibili all'interno del programma.

Prendi l'alieno funzionerà nel seguente modo:



Definito il diagramma di flusso si rende necessario implementare le funzioni che realizzano i singoli blocchi, utilizzando le facilitazioni che offre la libreria Pygame.

Actor

Gli Actor sono *oggetti* che devono apparire nel gioco. Per oggetto si intende un elemento software, appartenente ad una *classe* di oggetti, al cui interno mantiene una serie di *attributi* (informazioni) che lo caratterizzano. Contiene inoltre funzionalità, dette *metodi*, che definiscono comportamenti o azioni da poter realizzare con l'oggetto stesso. Per i dettagli sul concetto

teorico di oggetto, all'interno del paradigma di programmazione ad oggetti, si consigliano altri manuali. In questo contesto verranno utilizzati oggetti e classi considerando scontato il loro funzionamento.

Pygame mette a disposizione una classe di oggetti denominata **Actor**, in tal modo sarà molto facile introdurre nel gioco singoli elementi, associandogli un'immagine ed applicando loro tutte le funzionalità necessarie.

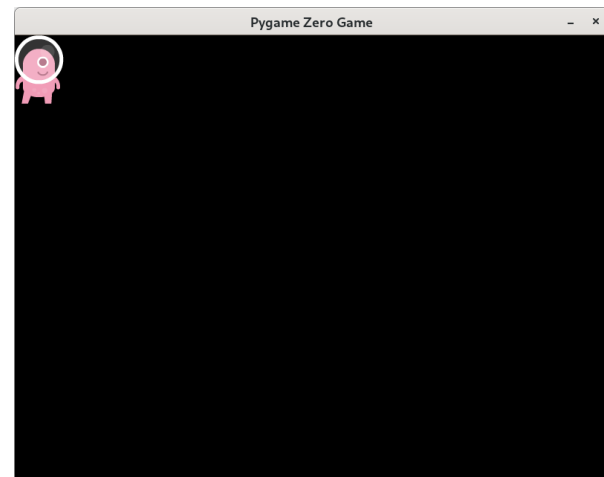
All'interno della cartella che contiene lo script, bisogna creare una cartella che si chiama *images*, in tal modo pygame andrà a prendere l'immagine associata al singolo attore, facendo attenzione che il nome del file abbia solo caratteri minuscoli. La sintassi per mettere in relazione immagine ed attore è la seguente:

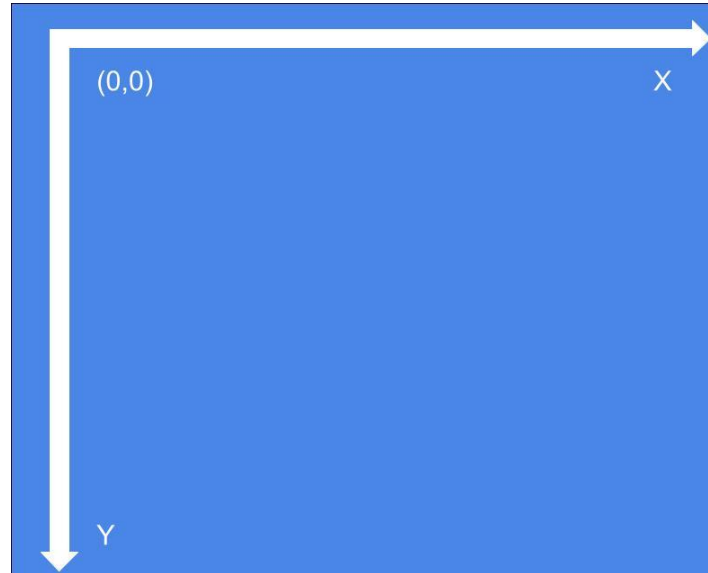
```
alien = Actor("alien")

def draw():
    screen.clear()
    alien.draw()
```

In tal modo la variabile **alien** risulta un oggetto di tipo **Actor** al quale pygame associa l'immagine *alien.png* presente nella cartella *images*. All'interno della funzione *draw()* viene inoltre richiamato il metodo *draw()* associato all'oggetto *alien*. Come si può notare dalla sintassi, il punto separa il nome dell'oggetto dal metodo che si vuole richiamare. In tal modo si otterrà la schermata accanto.

Si noti che l'immagine dell'oggetto viene collocata in alto a sinistra, perchè il punto di coordinate (0,0) è in alto a sinistra ed il semiasse positivo delle Y scorre verso il basso. Per posizionare un oggetto quindi bisogna collocarlo in un sistema cartesiano di questo tipo:





Il gioco prevede che l'alieno si posizioni in un punto diverso dello schermo ad ogni click, quindi sarà necessario costruire una funzione che generi coordinate casuali da associare ad Alien.

```
from random import randint

alien = Actor("alien")

✓ def draw():
    |     screen.clear()
    |     alien.draw()

✓ def place_alien():
    |     alien.x = randint(10,800)
    |     alien.y = randint(10,600)
```

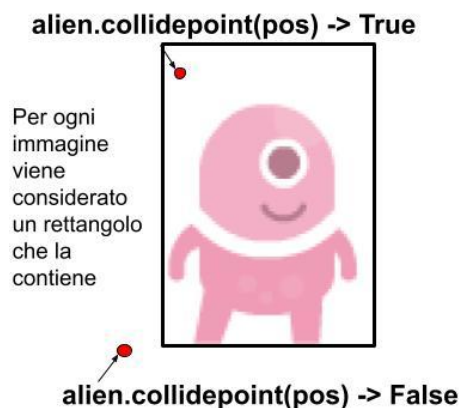
La funzione **randint()** genera un numero casuale tra gli interi che vengono passati, in questo caso la x varierà tra 10 ed 800 e la y tra 10 e 600. Si noti inoltre che per usare tale funzione bisogna richiamare il modulo **random**, importando **randint**.

Click del mouse

Per acquisire il click dell'utente esiste una funzione built-in messa a disposizione da Pygame Zero: **on_mouse_down(pos)**. Il parametro *pos* che prende in input è la posizione dell'oggetto, sarà sempre pygame ad occuparsi di prelevarla. Non resterà quindi che acquisire l'input e seguire la logica in funzione del diagramma di flusso costruito in precedenza.

In "Prendi l'alieno" si deve valutare la posizione del click, confrontandola con la posizione dell'alieno. Per realizzare questo confronto esiste un'altra funzione built-in appartenente all'oggetto *Actor* che prende in input la posizione dell'alieno e verifica la sovrapposizione con la posizione del mouse: **alien.collidepoint(pos)**. Si tenga presente che un **Actor**, la cui immagine è salvata nella cartella *images*, conserva diverse caratteristiche, tra cui la posizione salvata come coppia (x,y) nella variabile **pos** già citata.

Per colpire l'alieno il mouse deve cadere in una posizione interna al rettangolo che include l'immagine, anche se questa è trasparente lungo i bordi.



Per implementare la logica scelta si deve utilizzare il costrutto di selezione all'interno della funzione che definisce il comportamento da tenere al click del mouse:

```
def on_mouse_down(pos):  
    if alien.collidepoint(pos):  
        print("Bel colpo!!")  
        place_alien()  
    else:  
        print("Mancato!")  
        quit()
```


A questo punto non resta che posizionare l'alieno utilizzando la funzione creata e dare inizio al gioco. Lo script complessivo sarà il seguente:

```
3 |
4 | from random import randint
5 |
6 | alien = Actor("alien")
7 |
8 | def draw():
9 |     screen.clear()
10 |    alien.draw()
11 |
12 | def place_alien():
13 |     alien.x = randint(10,800)
14 |     alien.y = randint(10,600)
15 |
16 |
17 | def on_mouse_down(pos):
18 |     if alien.collidepoint(pos):
19 |         print("Bel colpo!!")
20 |         place_alien()
21 |     else:
22 |         print("Mancato!")
23 |         quit()
24 |
25 | place_alien()
26 |
```

Ordine nello script

Python interpreta il codice, questo vuol dire che tutto deve essere scritto in ordine di “apparizione”, altrimenti potrebbe non riconoscere alcuni elementi. Un buon ordine degli elementi potrebbe essere il seguente:

1. Importare le librerie esterne.
2. Dichiarare le variabili, gli *Actor* e tutti gli elementi necessari.

3. Dichiarazioni di funzioni built-in e personalizzate.
4. Funzione principale.

Messaggi in sovrapposizione

L'aspetto del gioco può ovviamente essere modificato, integrato ed ampliato grazie a funzioni built-in articolate e a disposizione del programmatore. Si consideri ad esempio la possibilità di personalizzare la grandezza dello schema di gioco oppure visualizzare scritte in sovrapposizione e non solo nel terminale.

Prima di procedere ad altre modifiche è possibile personalizzare il campo di gioco. Per farlo si può agire sui parametri **WIDTH** e **HEIGHT**, parametri interni al sistema attraverso cui definire larghezza ed altezza del quadro e che vanno inseriti prima di tutte le funzioni:

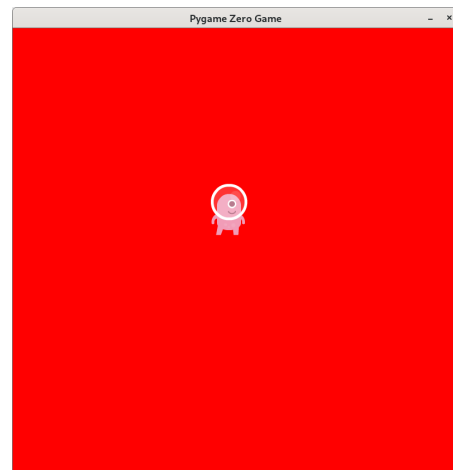
```
WIDTH = 800  
HEIGHT = 800
```

Allo stesso modo è possibile modificare la posizione dell'alieno:

```
def place_alien():  
    alien.x = randint(10,WIDTH-10)  
    alien.y = randint(10,HEIGHT-10)
```

Per personalizzare lo sfondo bisogna agire sulla funzione *draw()*, ad esempio impostare lo sfondo rosso:

```
def draw():  
    screen.clear()  
    screen.fill("red")  
    alien.draw()
```



Ultima modifica proposta è restituire un output testuale nel campo di gioco. Per ottenere questo risultato bisogna dichiarare una variabile di tipo stringa da visualizzare nell'area di gioco ed aggiornarla ogni volta che si clicca sull'alieno. Bisogna quindi procedere con tre modifiche:

1. Dichiarazione di una variabile di tipo stringa, inizialmente vuota. Ipotizziamo si chiami **msg**.
2. In **draw()** va impostato il colore, la posizione ed eventuali altri parametri relativi alla visualizzazione di **msg**.
3. In **on_mouse_down(pos)** va richiamata la variabile *msg*, attraverso l'uso del parametro **global**. Questo passaggio è fondamentale perchè altrimenti la variabile *msg* non sarà visibile all'interno della funzione.

```
WIDTH = 800
HEIGHT = 800

msg = ""

def draw():
    screen.clear()
    screen.fill("red")
    alien.draw()
    screen.draw.text(msg, topleft=(WIDTH/2-30,10), color="white", fontsize=32)

def place_alien():
    alien.x = randint(10,WIDTH-10)
    alien.y = randint(10,HEIGHT-10)

def on_mouse_down(pos):
    global msg
    if alien.collidepoint(pos):
        msg = "Bel colpo!!"
        print(msg)

        place_alien()
    else:
        msg = "mancato!"
        print(msg)
        quit()

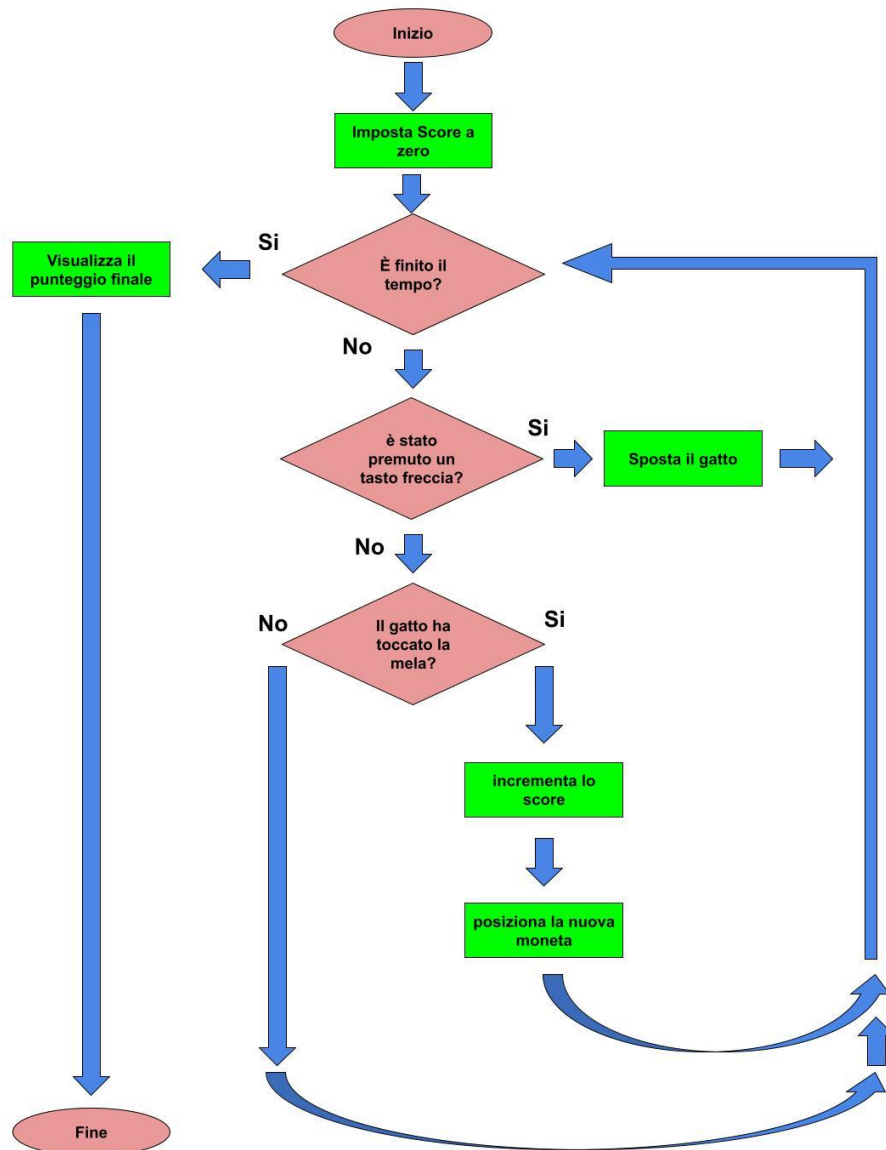
place_alien()
```

Esercitazione

- Si provi a salvare il numero di alieni colpiti e lo si visualizzi nella finestra.
- Si provi a cambiare sfondo e grandezza della finestra.

Mangia le monete

Si realizzi adesso un gioco in cui un gatto accumula punti mangiando monete d'oro. Il gioco finisce dopo 10 secondi ed il punteggio finale deve essere visto nella schermata finale. Un possibile diagramma di flusso è il seguente:



Variabili

Rispettando l'ordine con cui devono essere impostate le informazioni si deve introdurre la variabile **score**, inizializzandola a zero e la variabile **game_over** inizializzata a *False* (variabile di tipo booleano). Quando il tempo sarà scaduto *game_over* diventerà *True* ed il gioco potrà terminare.

```
3  WIDTH = 800
4  HEIGHT = 800
5
6  score = 0
7
8  game_over = False
9
```

Actor

A differenza del gioco precedente, in questo caso avremo due oggetti di tipo *Actor*, uno sarà usato dall'utente e l'altro è quello che dovrà essere catturato per accumulare punti. Dopo aver creato gli oggetti, verranno posizionati inizialmente in modo fisso, in due posizioni diverse del campo di gioco.

```
9
10 coin = Actor("coin")
11 coin.pos = 200,200
12
13 cat = Actor("gatto1")
14 cat.pos = 100,200
15
```

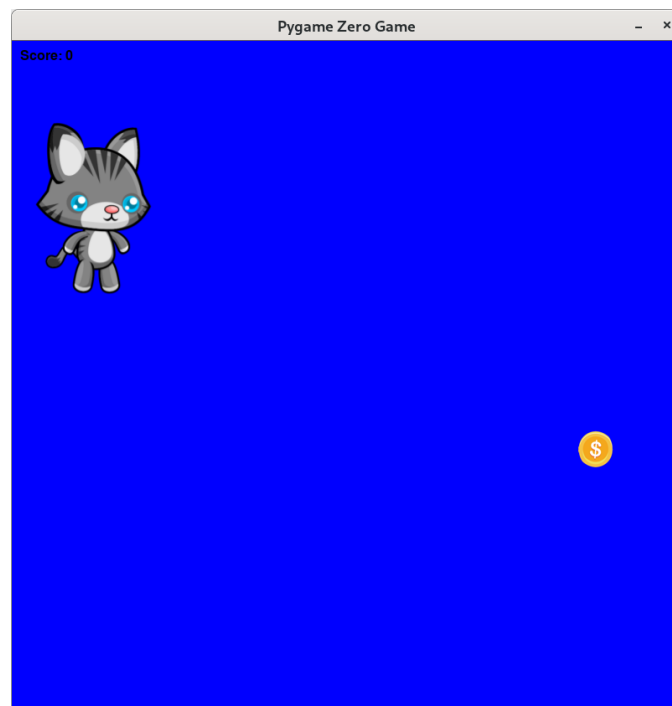
Nella funzione *draw()* devono essere visualizzati gli attori e lo score, posizionandolo in alto a sinistra nello schermo.

```
..
20 def draw():
21
22     screen.clear()
23     screen.fill("blue")
24     coin.draw()
25     cat.draw()
26
27     screen.draw.text("Score: " + str(score), color="black", topleft=(10,10))
28
```

Oltre la funzione *draw* bisogna implementare altre funzioni per la visualizzazione degli oggetti, in particolare quella che posiziona una moneta in modo casuale sullo schermo, come per il gioco precedentemente creato. Ricordiamo che per introdurre un elemento casuale è necessario importare il modulo **random**.

```
32
33 def place_coin():
34     coin.x = randint(20, (WIDTH-20))
35     coin.y = randint(20, (HEIGHT-20))
36
```

A questo punto lo schermo di gioco sarà il seguente:



Se tutto ha funzionato correttamente è possibile procedere con la gestione degli altri blocchi presenti nel diagramma di flusso. Risulta necessario quindi introdurre altre funzioni, per gestire l'evoluzione e la chiusura del gioco. Per non rischiare di dimenticare qualche elemento presente nel diagramma di flusso, risulta comodo dichiarare le funzioni senza implementarle necessariamente, attraverso l'uso della parola chiave **pass**. L'uso di tale parametro permette all'interprete di eseguire il codice senza generare errore, perchè se dovesse incontrare una funzione vuota, senza alcuna istruzione, genererebbe un' improvvisa interruzione dell'esecuzione.

```
15
16 def time_up():
17     pass
18
19 def update():
20     pass
21
```

Gestione del tempo

Il diagramma di flusso prevede la chiusura del gioco dopo 10 secondi. Per gestire il tempo bisogna introdurre una funzione built-in, che chiamerà la funzione ***time_up()*** la quale cambierà lo stato della variabile ***game_over*** dichiarata inizialmente come *False*. *time_up()* deve essere dichiarata, per una questione di ordine del codice, tra le funzioni subito dopo l'introduzione delle variabili.

time_up() richiama la variabile *game_over* come globale e poi la imposta a *True*.

```
16
17 def time_up():
18     global game_over
19     game_over = True
20
```

Il metodo ***schedule*** del modulo ***clock***, nativo di python, prende in input due parametri: la funzione da avviare ed i secondi dopo i quali avviarla. Quando il gioco viene seguito, la prima istruzione genera un timer di 10 secondi, al termine del quale viene lanciata la funzione *time_up*, che converte il valore di *game_over* da *False* a *True*. Si noti che l'istruzione che avvia *clock.schedule()* sarà la prima istruzione dell'algoritmo principale.

```
60
61 clock.schedule(time_up, 10.0)
62
63 place_coin()
64
```

Variabili locali e variabili globali

All'interno di un programma le variabili possono avere diversi tipi di visibilità. Una variabile si definisce ***globale*** quando è accessibile da tutte le funzioni incluse nel programma e da tutte le istruzioni che lo compongono. Si definisce ***locale***, invece, quando è accessibile solo da un determinato blocco di istruzioni. In Python, un blocco è identificato dal gruppo di istruzioni che

hanno lo stesso livello di intenzione. Si intuisce quindi, che una variabile globale deve essere dichiarata nella parte di codice più esterna, mentre quelle locali vanno dichiarate nei blocchi di codice più interni. Per essere accessibile una variabile globale all'interno di una funzione va utilizzata la parola chiave **Global**, così come visto nella funzione *time_up()*.

Game over

Quando *time_up()* cambia il valore contenuto in *game_over* che deve terminare l'esecuzione del gioco. Un modo semplice per farlo è cambiare la schermata di gioco, quindi bisogna agire sulla funzione *draw()*, inserendo la verifica dello stato di *game_over*. Poichè *draw()* viene eseguito ciclicamente, molte volte al secondo, nel momento in cui cambia lo stato di *game_over* è possibile cambiare lo sfondo *screen*, visualizzando il punteggio raggiunto.

```
22 def draw():
23
24     screen.clear()
25     screen.fill("blue")
26     coin.draw()
27     cat.draw()
28
29     screen.draw.text("Score: " + str(score), color="black", topleft=(10,10))
30
31     if game_over:
32         screen.fill("pink")
33         screen.draw.text("Punteggio finale: " + str(score), topleft=(10,10), fontsize = 60)
34
```

update()

L'ultima funzione da introdurre è la funzione built-in *update()* che viene eseguita da Pygame 60 volte al secondo automaticamente, senza quindi la necessità di richiamarla. Nel nostro caso dovrà occuparsi di due principali aspetti:

1. Collezionare le monete ed aggiornare la variabile *score*.
2. Ricevere input dalla tastiera per far spostare il gatto nel quadro.

Per incrementare il punteggio bisogna utilizzare all'interno di *update()* la variabile *coin* attraverso il parametro *global*. Ad ogni esecuzione, *update()* verifica la collisione tra i due oggetti attraverso il metodo *colliderect* applicato all'oggetto *cat*, il cui risultato (*True* o *False*) viene salvato in *coin_collector*. Per aggiornare lo score basterà verificare lo stato di *coin_collector* attraverso un blocco selezione. Dopo aver incrementato il punteggio bisogna posizionare un nuovo oggetto di tipo *Coin* all'interno del campo di gioco.


```
40
41 def update():
42     global score
43     coin_collected = cat.collidect(coin)
44
45     if coin_collected:
46         score = score + 10
47         place_coin()
48
49
50
51
```

si noti come *collidect* viene applicato all'oggetto *cat*, passandogli come parametro l'oggetto *coin*.

Input da tastiera


L'ultimo blocco istruzione del diagramma di flusso da realizzare riguarda il movimento del gatto, attraverso l'uso della tastiera, un oggetto built-in che ha come attributi i tasti della tastiera reale. Ogni attributo è rappresentato da un valore booleano inizialmente *False*, la pressione del relativo tasto lo rende *True*. Per spostare il gatto nel campo di gioco risultano comode le frecce del cursore, che corrispondono agli attributi **left**, **right**, **up** e **down** dell'oggetto **keyboard**. La verifica dello stato delle frecce va inserito nella funzione *update()*, così da verificare ad ogni ciclo (sessanta volte al secondo) se l'utente ha usato uno dei 4 tasti. In tal modo è possibile modificare la posizione del gatto sull'asse X o Y in relazione alla freccia selezionata dall'utente, incrementando o decurtando la posizione del gatto, ad esempio, di 2 pixel alla volta.

```
51
52     if keyboard.left:
53         cat.x = cat.x - 2
54     elif keyboard.right:
55         cat.x = cat.x + 2
56     elif keyboard.up:
57         cat.y = cat.y - 2
58     elif keyboard.down:
59         cat.y = cat.y + 2
60
61
```

Esercitazione

Si provi a modificare il programma appena realizzato, agendo su:

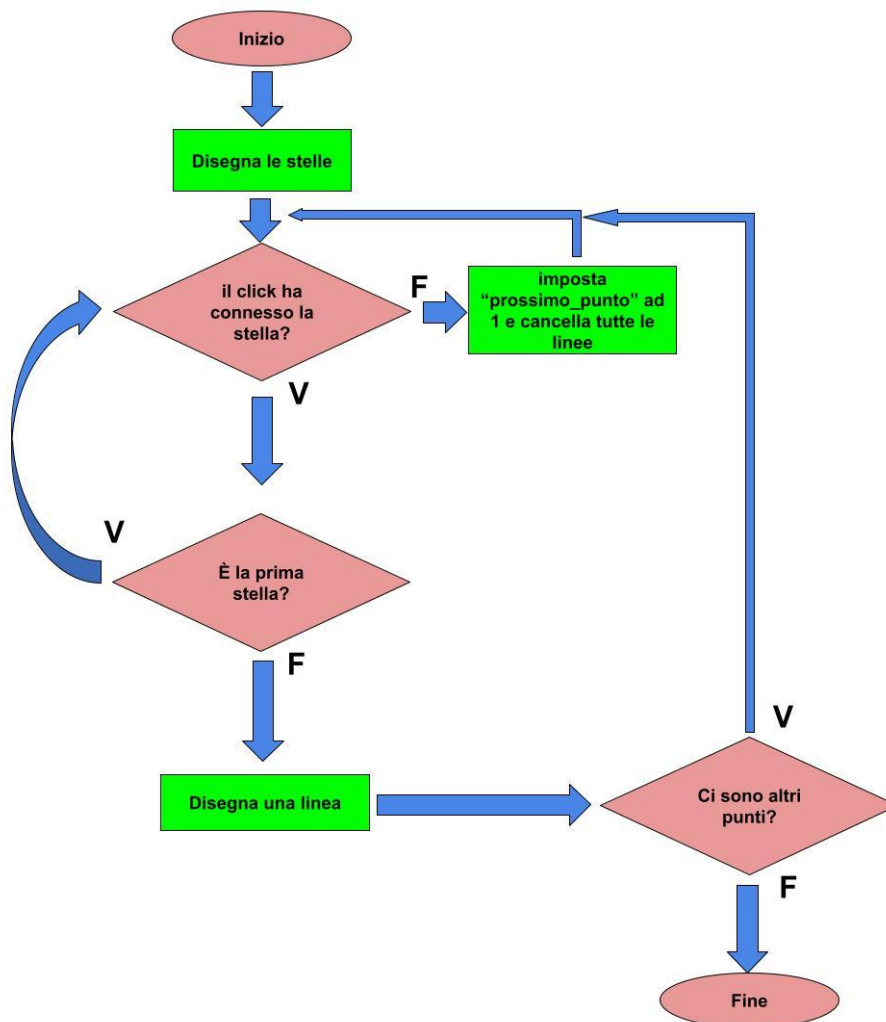
- Durata della partita
- Numero di oggetti, oltre la moneta ci potrebbero essere più oggetti con punteggi diversi.

- 
- Sfondo, grandezza e colore. Si tenga presente che il colore può essere assegnato tramite valori numerici di tipo RGB.
 - Rendere il gatto più veloce.
 - Si modifichi il modo di visualizzare il punteggio finale.

Unisci le stelle

Il gioco consiste nel selezionare in ordine corretto le stelle disposte in modo casuale sullo schermo, il giocatore deve cliccare nell'ordine indicato dal computer altrimenti si perde la partita. Ad ogni click deve apparire una linea che unisce la stella precedente alla stella selezionata.

All'avvio del gioco appaiono otto stelle in posizioni casuali, identificate da un numero progressivo, il giocatore deve cliccare sulle stelle rispettando l'ordine dei numeri, nel più breve tempo possibile. Il diagramma di flusso che lo rappresenta è il seguente:



Variabili di tipo lista

Rispettando l'ordine delle istruzioni da impartire, la prima cosa da fare è importare le librerie necessarie e definire le variabili da utilizzare. Dalla descrizione iniziale si evince che servirà la libreria **random** che dovranno essere utilizzate variabili strutturate di tipo lista:

- **stelle**, che dovrà contenere tutti gli oggetti di tipo *Actor* (le stelle).
- **linee**, che dovrà contenere tutti gli oggetti di tipo *Linea*.
- **prossima_stella**, utilizzata per tenere traccia delle stelle cliccate.

```
1  from random import randint
2
3  WIDTH = 400
4  HEIGHT = 400
5
6  stelle = []
7  linee = []
8
9  prossima_stella = 0
10
```

Stelle e **linee** sono variabili capaci di contenere al proprio interno più elementi, per tale motivo vengono dichiarate con l'uso di parentesi quadre. In tal modo l'interprete python sarà che conterranno più elementi, ciascuno dei quali corrisponde ad un indice preciso in ordine progressivo.

Una variabile di tipo *lista* può contenere elementi di tipo diverso, sempre ordinati in modo progressivo in relazione all'inserimento. Può essere, quindi, interpretata come una cassetiera ed ogni cassetto contiene un valore diverso.

Ciclo di for

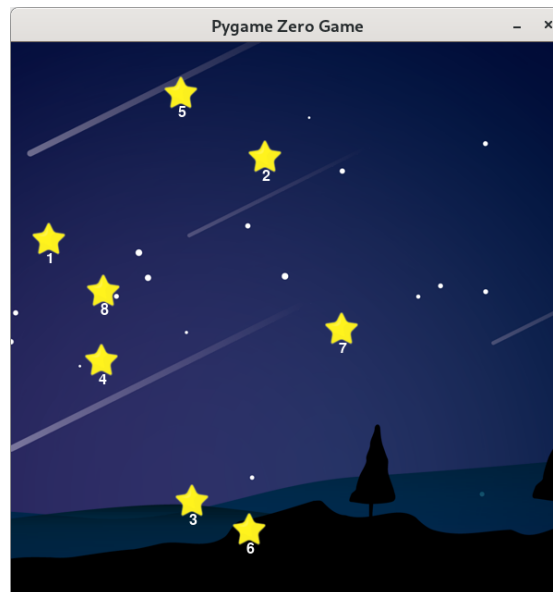
Il primo blocco di istruzioni prevede il disegno di tutte le stelle, ognuna delle quali è un oggetto *Actor* di tipo stella. In un caso del genere è necessario introdurre un'istruzione ciclica che permette di "riempire" la lista *stelle* di oggetti, utilizzando la funzione `range(n,m)` che genera una sequenza di numeri che vanno da *n* (incluso) ad *m* (escluso).

```
10
11  for stella in range(0,8):
12      actor = Actor("stella")
13      actor.pos = (randint(20, WIDTH -20),randint(20, HEIGHT-20))
14      stelle.append(actor)
15
```

Si noti che il ciclo di *for* è un blocco di codice all'interno del quale la variabile *stella* varia da zero a sette, realizzando otto ripetizioni delle istruzioni successive. La variabile *actor*, ad ogni iterazione, crea un oggetto di tipo Actor con l'immagine "stella", genera una posizione casuale nel campo di gioco ed attraverso il metodo *append()* aggiunge l'oggetto creato alla lista *stelle*.

Per disegnare le stelle bisogna applicare lo stesso tipo di ciclo alla funzione *draw()*, in modo da posizionare tutte le stelle.

```
15
16 def draw():
17     screen.blit("cielo",(0,0))
18     num = 1
19     for stella in stelle:
20         screen.draw.text(str(num),(stella.pos[0]-3,stella.pos[1] +14))
21         stella.draw()
22         num = num + 1
```



Si noti che:

- Si è scelto di utilizzare un'immagine come sfondo attraverso il metodo **blit()** dell'oggetto *Screen*, questo prende in input il nome del file immagine salvato nella cartella *images* e la posizione (x,y) in cui collocare l'angolo in alto a sinistra dell'immagine stessa.
- La variabile *num* permette di ordinare le stelle, in modo da poter scrivere l'indice numerico al di sotto dell'immagine che appare nel campo di gioco.
- Il ciclo di *for* associa alla variabile *stella* un oggetto alla volta prelevato dalla lista *stelle*. Ad ogni iterazione del ciclo viene scritto sullo schermo il numero salvato nella variabile

num (dopo averlo trasformato da intero a stringa attraverso la funzione *str()*). Al numero da scrivere viene attribuita la posizione relativa alla Stella a cui fa riferimento, traslata di 3 pixel lungo l'asse X e 14 pixel lungo l'asse Y. La traslazione è necessaria affinché il numero non sia completamente sovrapposto alla Stella, ma risulti centrato in basso ad essa. Alla fine del ciclo si incrementa la variabile *num*, così da poter disegnare un altro oggetto stella ed arrivare alla fine della lista.

Disegnare una linea

In coda alla funzione *draw()* bisogna aggiungere un ciclo che disegna tutte le linee che congiungono in modo corretto le stelle. Bisogna notare che tale ciclo preleva le linee da disegnare dalla lista *linee* inizialmente vuota, ma che viene popolata man mano che il giocatore clicca in modo corretto sulle stelle.

```
23 |  
24 |     for linea in linee:  
25 |         screen.draw.line(linea[0], linea[1],(100,0,0))  
26 |  
27 |
```

Si noti che la funzione *draw.line()* appartiene all'oggetto *screen* e prende in input tre parametri, il punto iniziale, il punto finale ed il colore della linea, inteso come terna RGB. in questo caso quando si clicca sulla prima stella non deve essere disegnata nessuna linea, che invece partirà dalla prima stella nel momento in cui verrà selezionata la stella successiva. Si noti inoltre che ogni elemento della variabile *linee*, è composto di due elementi accessibili attraverso le parentesi quadre. Il primo elemento corrisponde all'indice 0 ed il secondo elemento all'indice 1.

A questo punto bisogna codificare cosa accade quando viene cliccato il tasto sinistro del mouse. Il diagramma di flusso descrive il comportamento in modo preciso con la funzione *on_mouse_down()*: se la stella selezionata è quella corretta bisogna aggiungere una linea alla lista *linee*, in caso contrario devono essere cancellate tutte le linee appartenenti a quella lista e inizializzare a 0 la variabile *prossima_stella*.

```

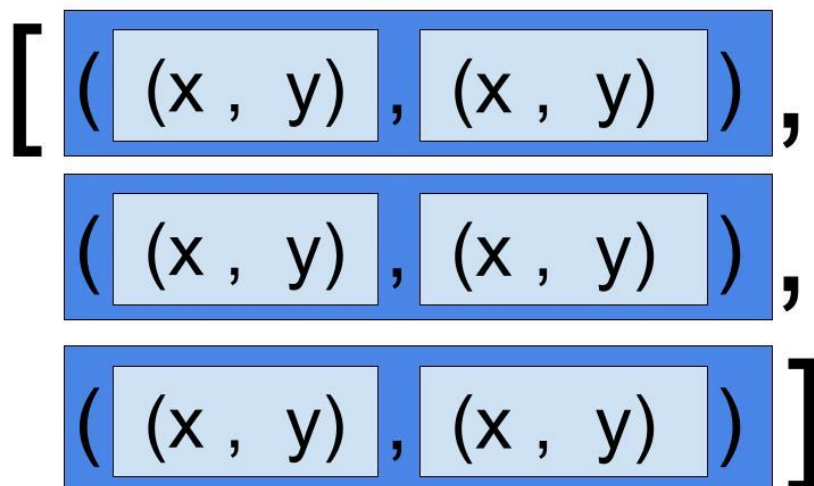
28 def on_mouse_down(pos):
29
30     global prossima_stella
31     global linee
32
33     if stelle[prossima_stella].collidepoint(pos):
34         if prossima_stella:
35             linee.append((stelle[prossima_stella-1].pos, stelle[prossima_stella].pos))
36             prossima_stella = prossima_stella + 1
37
38     else:
39         linee = []
40         prossima_stella = 0

```

Si noti la presenza di due strutture di selezione innestate. Questo perché se la posizione (*pos*) del puntatore del mouse è all'interno del riquadro della stella con indice *prossima_stella*, bisogna verificare anche che *prossima_stella* non sia la prima (indice 0); in questo caso, infatti, non deve essere aggiunta nessuna linea.

Tuple

È utile notare che ogni elemento della variabile strutturata *linee*, è a sua volta un elemento strutturato. Infatti, osservando il codice, si evince che l'elemento da aggiungere alla lista è una coppia formata dalle posizioni del punto iniziale della linea e del punto finale, queste posizioni a loro volta risultano essere composte da due elementi: X e Y. Emerge, quindi, una struttura molto complessa che potremmo rappresentare in questo modo:



Si noti che con le parentesi quadre si identifica una lista, cioè un insieme di oggetti **modificabili** e potenzialmente di vari tipi. In questo caso gli oggetti appartenenti alla lista principale sono gli

estremi delle linee da disegnare. Ogni elemento della lista, identificato dalle parentesi tonde, è invece un tipo di struttura dati che prende il nome di **tupla**. Questo tipo di struttura, a differenza dalla *lista*, **non è modificabile** perché quando viene creato un oggetto di questo tipo, non è possibile apporre modifiche. Per questa sua caratteristica è facile intuire che una tupla risulta particolarmente adatta a rappresentare le posizioni fisse all'interno di un piano cartesiano. Nel caso specifico ogni elemento della lista *linee* è una tupla al cui interno sono identificate 2 *tuple*, la prima rappresenta la coppia (x,y) relativa al punto iniziale della linea e la seconda relativa alla coppia (x,y), coordinate del punto finale. Infine, per accedere agli elementi di una lista o di una tupla, si procede con la stessa sintassi: si utilizza un indice tra le parentesi quadre per identificare l'elemento desiderato.

Il risultato finale è rappresentato nella seguente figura.



Gestione del tempo con time()

Lo scopo del gioco descritto inizialmente prevede il calcolo del tempo impiegato a congiungere le stelle. Per ottenere questo risultato bisogna introdurre il modulo **Time**, che permette di calcolare un tempo assoluto a partire da un tempo iniziale, che ogni sistema operativo determina. A partire da quel tempo è possibile calcolare un tempo relativo di inizio e fine gioco.

Bisogna quindi:

- importare la libreria *Time*.

- introdurre 3 variabili utili a contenere il tempo iniziale, finale e totale.
- introdurre le opportune modifiche alle funzioni coinvolte.

```
1  from random import randint
2  from time import time
3
4  WIDTH = 600
5  HEIGHT = 600
6
7  stelle = []
8  linee = []
9
10 numero_stelle = 8
11 prossima_stella = 0
12
13 tempo_iniziale = 0
14 tempo_totale = 0
15 tempo_finale = 0
```

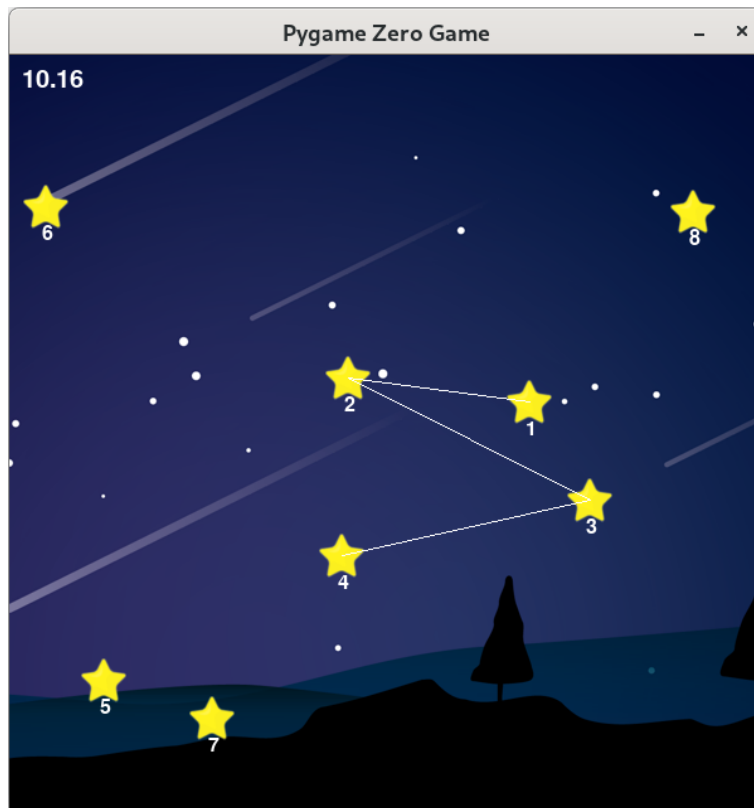
Si noti che è stato introdotto anche il numero di stelle sotto forma di variabile **numero_stelle**, in questo modo sarà molto più facile controllare in tutti i segmenti del codice il numero massimo di stelle da visualizzare.

Nella funzione *draw()* verrà calcolato il tempo e visualizzato in alto a sinistra. Il tempo viene aggiornato appena si seleziona una stella.

```
36     for linea in linee:
37         screen.draw.line(linea[0], linea[1],(255,255,255))
38
39     if prossima_stella < numero_stelle:
40         tempo_totale = time() - tempo_iniziale
41         # screen.draw.text(str(tempo_totale), (10, 10) ,fontsize=30)
42         screen.draw.text(str(round(tempo_totale, 2)), (10, 10) ,fontsize=30)
43     else:
44         # screen.draw.text(str(tempo_totale), (10, 10) ,fontsize=30)
45         screen.draw.text(str(round(tempo_totale, 2)), (10, 10) ,fontsize=30)
46
```

La funzione *time()* restituisce il numero di millisecondi a partire dal tempo *t0*, così come illustrato in precedenza. Si noti, inoltre, che il tempo totale viene aggiornato soltanto se non si è selezionata l'ultima stella. Le righe di codice commentate permettono di visualizzare il tempo totale con tutte le cifre significative, ma attraverso la funzione *round()* è possibile arrotondare il

valore reale che gli viene passato, indicando come secondo parametro il numero di cifre significative da visualizzare, in questo caso due.



Fase di test

Effettuare un test dello script è sempre una buona norma, provando ad inserire ogni forma possibile di input. In questo caso, dopo aver selezionato tutte le stelle, se si clicca su un punto qualsiasi dell'area di gioco, viene restituito al terminale l'errore riportato di seguito. Questo vuol dire che si è provato ad accedere ad un'area di memoria eccedente la lista *stelle*, errore molto comune quando si opera con i cicli.

```
File "unisciStelle-2.py", line 51, in on_mouse_down
    if stelle[prossima_stella].collidepoint(pos):
IndexError: list index out of range
```

Per risolvere questo problema è sufficiente modificare la funzione *on_mouse_down()*, inserendo un blocco che controlla il numero di stelle che sono state già selezionate.

```
46
47 def on_mouse_down(pos):
48
49     global prossima_stella
50     global linee
51
52     if prossima_stella < numero_stelle:
53         if stelle[prossima_stella].collidepoint(pos):
54             if prossima_stella:
55                 linee.append((stelle[prossima_stella-1].pos, stelle[prossima_stella].pos))
56                 prossima_stella = prossima_stella + 1
57             else:
58                 linee = []
59                 prossima_stella = 0
60
61
```

Esercitazione

Sulla base dell'ultima versione del gioco si provi a:

- Aggiungere più stelle.
- Fare in modo che dopo aver selezionato l'ultima Stella, il computer propone lo stesso gioco con 2 stelle in più.