

Soluzioni degli esercizi

Queste soluzioni sono proposte soprattutto per favorire un'acquisizione progressiva delle conoscenze. Bisogna partire dall'assunto che esse *non* siano le uniche o le migliori soluzioni. Prima di studiare queste soluzioni, ognuno deve cercare in autonomia le *proprie*, che potranno anche essere molto diverse da quelle proposte. Alcune delle soluzioni seguenti potrebbero essere incomplete e presentare solo alcune idee per risolvere gli aspetti più critici del problema.

In queste proposte di soluzione noterete che i nomi delle variabili, i commenti ecc. sono in inglese. Un suggerimento è quello di provare a operare sul codice per esempio “*traducendolo*” in italiano in modo da riflettere sulla sua logica e il suo contenuto.

Esercizi capitolo 7 - Relazioni tra classi

Punti e segmenti

```
class Point:

    def __init__(self, x: float, y: float):
        self._x = x
        self._y = y

    def distance(self, p: "Point") -> float:
        return ((self._x - p._x) ** 2 + (self._y - p._y) ** 2) ** 0.5
```

```
class LineSegment:

    def __init__(self, a: Point, b: Point):
        self._a = a
        self._b = b

    def length(self) -> float:
        return self._a.distance(self._b)
```

Rana nell'arena

```
class Frog(Actor): # ...
    def move(self, arena):
        if arena.collisions():
            self._x, self._y = self._x0, self._y0

        keys = arena.current_keys()
```

```

if "a" in keys and self._count == 0:
    self._count = self._steps
    self._dx, self._dy = -self._speed, 0
# ...

if self._count > 0:
    self._count -= 1
    x = self._x + self._dx
    y = self._y + self._dy

```

▶ https://fondinfo.github.io/play/?p32_frog.py

La rana comincia il salto, solo dopo aver completato quello precedente: `self._count == 0`. Avanza solo se è in salto: `self._count > 0`. A ogni frame, il contatore è decrementato.

Rana sui tronchi

I tronchi si muovono in maniera del tutto simile ai veicoli di *Frogger*. Siccome la rana deve conoscere velocità orizzontale dei tronchi su cui si appoggia, questi forniscono un apposito metodo `getter`.

Per la rana, è interessante la gestione il trascinamento orizzontale. Se la rana atterra su un tronco (collisione, con conteggio 0), allora memorizza la velocità del tronco come *drift*. Questa deriva orizzontale è applicata alla rana per ogni successivo frame, anche durante i salti. Se invece la rana atterra altrove (conteggio 0, ma senza collisione), il *drift* viene azzerato. Per il resto, il personaggio è simile a quello dell'esercizio precedente.

```

class Frog(Actor): # ...
    def move(self, arena):
        raft = False
        for other in arena.collisions():
            if isinstance(other, Raft) and self._count == 0:
                raft = True
                self._drift = other.speed()
                self._x += self._drift

        if self._count > 0:
            self._count -= 1
            self._x += self._dx + self._drift
            self._y += self._dy
        else:
            if not raft:
                self._drift = 0

```

▶ https://fondinfo.github.io/play/?p32_raft.py

Super Mario

Riprendiamo ancora la struttura dei giochi di animazione e iniziamo a creare i nostri nuovi personaggi. In questo caso si tratta di uno dei più famosi e longevi nella storia dei videogiochi? *Super Mario!* ¹

Ma da dove si comincia? Anche in questo caso, partiamo da un problema più semplice, in cui sono coinvolti solo due tipi di personaggi: *Wall*, le piattaforme su cui poggia Mario muovendosi o saltando, e *Mario*, che si muove a destra o sinistra e salta sulle varie piattaforme.

Wall. Si tratta di un personaggio del tutto immobile e passivo. In ogni caso, eredita (*implementa l'interfaccia Actor*). È utile nel gioco per la gestione delle collisioni.

```
class Wall(Actor):
    def __init__(self, pos, size):
        self._pos = pos
        self._size = size

    def move(self, arena):
        return

    def pos(self):
        return self._pos

    def size(self):
        return self._size

    def sprite(self):
        return None
```

Mario. Il protagonista si muove in base ai comandi dell'utente, forniti da tastiera.

Il salto si realizza semplicemente imponendo una certa velocità prefissata, verso l'alto. Ciò però è consentito solo se il personaggio è appoggiato sopra a una piattaforma.

Richiede un po' d'attenzione la gestione delle collisioni con una piattaforma, in particolare per scegliere il bordo a cui spostarsi.

```
class Mario(Actor): # ...
    def __init__(self):
        self._x, self._y = 0, 240
        self._w, self._h = 20, 20
        self._dx, self._dy = 2, 0
        self._speed, self._jump = 2, -8
        self._gravity = 0.25

    def move(self, arena):
        self._dx = 0
        if "ArrowRight" in arena.current_keys():
            self._dx = self._speed
```

¹<https://www.arcade-museum.com/Videogame/mario-bros>

```

elif "ArrowLeft" in arena.current_keys():
    self._dx = -self._speed
for other in arena.collisions():
    wall_x, wall_y = other.pos()
    wall_w, wall_h = other.size()
    # am I above the platform, going down?
    if self._y < wall_y and self._dy >= 0:
        self._y = wall_y - self._h # landed
        self._dy = 0
        if "ArrowUp" in arena.current_keys():
            self._dy = self._jump # jump
    # am I below the platform, going up?
    elif self._y+self._h > wall_y+wall_h and self._dy <= 0:
        self._y = wall_y + wall_h + 1
        self._dy = 0
    # ...

arena_w, arena_h = arena.size()
self._x = (self._x + self._dx) % arena_w
self._y += self._dy
self._dy += self._gravity

```

Analizziamo le proprietà di Mario, definite nel costruttore `__init__`:

- `self._x` e `self._y`: come in tutti i personaggi definiscono la posizione attuale
- `self._dx` e `self._dy`: direzione e spostamento (inizialmente è fermo)
- `self._w` e `self._h`: dimensione (larghezza e altezza)
- `self._speed`: costante velocità, spostamento assoluto in pixel
- `self._jump`: velocità iniziale del salto
- `self._gravity`: componente della forza di gravità (Mario salta)

`pos`, `size` e `sprite` sono metodi **getter** che restituiscono la posizione, la dimensione e la pozione dell'immagine di Mario all'interno del file che contiene l'immagine di tutti i personaggi del gioco.

La parte più importante e impegnativa è la gestione del movimento nel metodo `move`

Il gioco. Definiti i personaggi e l'arena questa prima bozza del gioco risulta semplice.

```

arena = Arena((640, 480))
arena.spawn(Wall((240, 350), (100, 40)))
arena.spawn(Wall((420, 250), (100, 40)))
arena.spawn(Wall((0, 460), (640, 20)))
arena.spawn(Mario())


g2d.init_canvas(arena.size())
g2d.main_loop(tick, 60)

```

Inseriamo i personaggi (`arena.spawn`), inizializziamo il canvas e richiamiamo ciclicamente il metodo `tick` che a ogni intervallo di tempo ripulisce il canvas, ridisegna i personaggi e comunica quali tasti sono stati premuti per gestire il movimento.

```
def tick():
    g2d.clear_canvas()
    for a in arena.actors():
        if a.sprite():
            g2d.draw_image("sprites.png", a.pos(),
                           a.sprite(), a.size())
        else:
            g2d.draw_rect(a.pos(), a.size())

    arena.tick(g2d.current_keys()) # Game logic
```

 https://fondinfo.github.io/play/?p32_mario.py

Bene, adesso puoi proseguire tu...

Scroll della vista

```
from p32_bounce import Ball, Arena

# arena & actors aren't aware of view
ARENA_W, ARENA_H = 500, 250
arena = Arena((ARENA_W, ARENA_H))
for _ in range(10):
    pos = randrange(ARENA_W - 20), randrange(ARENA_H - 20)
    arena.spawn(Ball(pos))

# view size is smaller than arena
VIEW_W, VIEW_H = 300, 200
BACKGROUND = "https://raw.githubusercontent.com/.../viewport.png"
view_x, view_y = 0, 0

def tick():
    global view_x, view_y
    keys = g2d.current_keys()
    if "ArrowUp" in keys:
        view_y = max(view_y - 10, 0)
    elif "ArrowRight" in keys:
        view_x = min(view_x + 10, ARENA_W - VIEW_W)
    elif "ArrowDown" in keys:
        view_y = min(view_y + 10, ARENA_H - VIEW_H)
    elif "ArrowLeft" in keys:
        view_x = max(view_x - 10, 0)

    # cut the visible background
    g2d.draw_image(BACKGROUND, (0, 0),
                   (view_x, view_y), (VIEW_W, VIEW_H))
    for a in arena.actors():
        x, y = a.pos()
        # translate sprites in view's coords
        g2d.draw_image("ball.png", (x - view_x, y - view_y))
    arena.tick()
```

▶ https://fondinfo.github.io/play/?p32_scroll.py

Lo scroll della vista può essere implementato totalmente nella funzione `tick`. Dell'immagine di sfondo viene ritagliata la porzione visibile con il metodo di `g2d` `draw_image`. Lo sfondo è semplicemente un'immagine disegnata prima delle altre. Alle posizioni dei personaggi si applica una semplice traslazione.

Nel codice dei personaggi, non si tiene affatto conto della vista e dello scroll; si continua a ragionare in termini dell'arena di gioco, anche se questa non è interamente visibile.

Pac-Man

Le coordinate dei muri sono organizzate in una lista di tuple (`walls`).

▶ https://fondinfo.github.io/play/?p32_pacmanmap.py

Nella lista delle collisioni fornita dall'arena, sono presenti anche casi di adiacenza, senza vera sovrapposizione. Se c'è una vera sovrapposizione in verticale, essa ostacola i movimenti in orizzontale del Pac-Man. Viceversa, una sovrapposizione in orizzontale ostacola i movimenti in verticale.

Si accettano comandi solo in direzioni non ostruite e solo in posizioni multiple di 8 (`tile`). Se, proseguendo nella sua direzione, nonostante tutto Pac-Man sbatte contro un muro, allora si ferma.

```
class PacMan(Actor): # ...
    def move(self, arena):
        path_l = path_r = path_u = path_d = True
        for other in arena.collisions():
            if isinstance(other, Wall):
                # wall can also be adjacent, w/o intersection
                ox, oy, ow, oh = other.pos() + other.size()
                if oy < self._y + self._h and self._y < oy + oh:
                    # | overlap, -- movement is obstacled
                    if self._x > ox:
                        path_l = False
                else:
                    path_r = False
                if ox < self._x + self._w and self._x < ox + ow:
                    # -- overlap, | movement is obstacled
                    if self._y > oy:
                        path_u = False
                else:
                    path_d = False

        if self._x % tile == 0 and self._y % tile == 0:
            # new direction, only if not leading against a wall
            keys = arena.current_keys()
            u, l, d, r = "wasd"
            if l in keys and path_l:
                self._dx, self._dy = -self._speed, 0
            elif r in keys and path_r:
                self._dx, self._dy = +self._speed, 0
            elif u in keys and path_u:
```

```

        self._dx, self._dy = 0, -self._speed
    elif d in keys and path_d:
        self._dx, self._dy = 0, +self._speed

    # if current direction is blocked, PacMan stops
    if (self._dx < 0 and not path_l or
        self._dx > 0 and not path_r or
        self._dy < 0 and not path_u or
        self._dy > 0 and not path_d):
        self._dx, self._dy = 0, 0

    arena_w, arena_h = arena.size()
    self._x = (self._x + self._dx) % arena_w
    self._y += self._dy

```

▶ https://fondinfo.github.io/play/?p32_pacman.py

Esercizi capitolo 8

Istogramma con barre orizzontali

```

W, H = 600, 400

values = []
txt = g2d.prompt("Val? ")
while txt:
    values.append(float(txt))
    txt = g2d.prompt("Val? ")

n, max_val = len(values), max(values)
for i, v in enumerate(values):
    pos = 0, i * H / n
    size = v * W / max_val, (H / n) - 1
    g2d.draw_rect(pos, size)

```

▶ https://fondinfo.github.io/play/?p41_histogram.py

La x e la h di ogni barra sono tutte uguali. La y è proporzionale a i . La w è proporzionale a v .

Risultati casuali

```

rolls = int(input("Rolls? "))
results = [0] * 11

for r in range(rolls):
    die1 = randint(1, 6)
    die2 = randint(1, 6)
    val = die1 + die2
    results[val - 2] += 1

```

```
for i, v in enumerate(results):
    #print(i + 2, v)
    print(f"{i + 2:2}", "=" * v) # f-string, string repetition
```

▶ https://fondinfo.github.io/play/?p41_dice.py

I risultati possibili vanno da 2 a 12. Usiamo una lista di 11 contatori. Contatore `results[0]`: quante volte esce il 2. Contatore `results[1]`: quante volte esce il 3, ecc.

Merge

```
def merge(a: list, b: list) -> list:
    result = []
    while a or b:
        if a and b:
            d = a if a[0] <= b[0] else b
        else:
            d = a if a else b
        result.append(d.pop(0))
    return result

data1 = [1, 4, 5, 7]
data2 = [2, 3, 6, 8, 9, 10]
merged = merge(data1, data2)
print(data1, data2, merged)
```

▶ https://fondinfo.github.io/play/?p41_merge.py

Non serve fare un ordinamento completo. Basta confrontare i primi valori delle due liste. Tra i due, viene estratto quello più piccolo. Sia `data1` che `data2` alla fine restano vuote.

Riempimento

```
def fill(values: list[int], pos: int):
    if values[pos] == 0:
        values[pos] = 1
        for d in (-1, 1):
            pos1 = pos + d
            while 0 <= pos1 < len(values) and values[pos1] == 0:
                values[pos1] = 1
                pos1 += d


data = [int(v) for v in "0022000000002000"]
fill(data, 9)
print(data)
```

▶ https://fondinfo.github.io/play/?p41_fill.py

Clamp di lista

```
def clamp(values: list[int], a: int, b: int):
    for i, v in enumerate(values):
        if v < a:
            values[i] = a
        if v > b:
            values[i] = b

data = [3, 4, 6, 7, 3, 5, 6, 12, 4]
clamp(data, 5, 10)
print(data)
```


 https://fondinfo.github.io/play/?p41_clamp.py

Shuffle

```
from random import randrange

def shuffle(values: list):
    for i, v in enumerate(values):
        j = randrange(len(values))
        values[i], values[j] = values[j], values[i]

data = list(range(10))
shuffle(data)
print(data)
```

 https://fondinfo.github.io/play/?p41_shuffle.py