

lezione15

November 13, 2023

1 Fondamenti di Programmazione

Andrea Sterbini

lezione 15 - 13 novembre 2023

2 RECAP: CLASSI e oggetti

- classi: attributi (di classe) e metodi
- istanze: attributi di istanza e metodi
- information hiding e “responsabilità” delle operazioni
- ereditarietà come meccanismo per estendere un tipo di oggetti

```
[1]: %load_ext nb_mypy
```

Version 1.0.5

2.1 Colori

- operazioni matematiche sui colori (somma, prodotto, ...)
- costruttore
- rappresentazione (`__repr__` e `__str__`)
- conversione in tripla RGB

```
[2]: import images
from random import randint

class Colore:
    white : 'Colore'
    black : 'Colore'
    red   : 'Colore'
    green : 'Colore'
    blue  : 'Colore'
    cyan  : 'Colore'
    purple: 'Colore'
    yellow: 'Colore'
    grey  : 'Colore'

    def __init__(self, R : float, G : float, B : float):
```

```

    "un colore contiene i tre canali R,G,B"
    self._R = R
    self._G = G
    self._B = B

    # può far comodo avere un secondo costruttore che genera colori casuali
    # per eseguirlo si scrive Colore.random()
    @classmethod
    def random(cls, m : int =0, M : int = 255) -> 'Colore' :    # -> Colore_
↪casuale
        "torna un colore casuale con i valori delle luminosità in [m .. M]"
        return cls(randint(m,M), randint(m,M), randint(m,M))

    def luminosità(self) -> float:    # luminosità del colore
        "calcolo la luminosità media di un pixel (senza badare se è un valore_
↪intero)"
        return (self._R + self._G + self._B)/3

    def __add__(self, other : 'Colore') -> 'Colore':
        "somma tra due colori"
        if not isinstance(other, Colore):
            raise ValueError("Il secondo addendo non è un Colore")
        return Colore(self._R + other._R, self._G + other._G, self._B + other.
↪_B)

    def __sub__(self, other : 'Colore') -> 'Colore':
        "somma tra due colori"
        if not isinstance(other, Colore):
            raise ValueError("Il secondo addendo non è un Colore")
        return Colore(self._R - other._R, self._G - other._G, self._B - other.
↪_B)

    def __mul__(self, k : float) -> 'Colore':
        "moltiplicazione di un colore per una costante"
        # FIXME: controllare che k sia un numero
        return Colore(self._R*k, self._G*k, self._B*k)

    def __truediv__(self, k : float) -> 'Colore':
        "divisione di un colore per una costante"
        # FIXME: controllare che k sia un numero != 0
        return Colore(self._R/k, self._G/k, self._B/k)

    def _asTriple(self) -> tuple[int, int, int]:
        "creo la tripla di interi tra 0 e 255 che serve per le immagini PNG"
        def bound(X):
            # solo quando devo creare un file PNG mi assicuro che il pixel sia_
↪intero nel range 0..255

```

```

        return max(0, min(255, round(X)))
    return bound(self._R), bound(self._G), bound(self._B)

    def __repr__(self) -> str :
        "stringa che deve essere visualizzata per stampare il colore"
        # uso una f-string equivalente a
        # return "Color(" + str(self._R) + ", " + str(self._G) + ", " +
↪str(self._B) + ")"
        return f"Colore({self._R}, {self._G}, {self._B})"

```

```

[3]: # solo dopo aver definito Colore posso aggiungere attributi che contengono un
↪Colore
Colore.white = Colore(255, 255, 255)
Colore.black = Colore( 0, 0, 0)
Colore.red = Colore(255, 0, 0)
Colore.green = Colore( 0, 255, 0)
Colore.blue = Colore( 0, 0, 255)
Colore.cyan = Colore( 0, 255, 255)
Colore.purple= Colore(255, 0, 255)
Colore.yellow= Colore(255, 255, 0)
Colore.grey = Colore.white/2

# Esempi
p1 = Colore(255, 0, 0)
p2 = Colore( 0,255, 0)
p3 = p2 + p1 # uso l'operatore somma tra due colori che ho definito sopra
p4 = p3 * 0.5 # uso l'operatore prodotto per una costante che ho definito
↪sopra

p5 = Colore(120, 34, 200)
p4

```

[3]: Colore(127.5, 127.5, 0.0)

3 Semplifichiamo le immagini

Per spostare le operazioni di disegno in classi separate ci servono solo le primitive di disegno minime:
- set_pixel e get_pixel - is_inside - __init__, __repr__, save e visualizza

```

[4]: import images
import math

class Immagine:

    def __init__(self, larghezza=None, altezza=None, sfondo=None,
↪filename=None):

```

```

"""posso creare una immagine in due modi:
- leggendola da un file PNG se passo il parametro filename
- fornendo dimensioni e colore di sfondo se non lo passo
"""
if filename:
    img = images.load(filename)
    # letta la immagine la converto in una matrice di Colore
    self._img = [ [ Colore(r,g,b) for r,g,b in riga ] for riga in img ]
↪]

    self._W = len(img[0])
    self._H = len(img)
else:
    # altrimenti creo una immagine monocoloro
    # FIXME: dovrei controllare che ci siano tutti e 3 gli altri
↪argomenti
    if sfondo is None:
        sfondo = Colore.black
    self._W = larghezza
    self._H = altezza
    self._img = [ [sfondo for _ in range(larghezza) ] for _ in
↪range(altezza) ]

def __repr__(self) -> str:
    "per stampare l'immagine ne mostro le dimensioni"
    return f"Immagine(larghezza={self._W},altezza={self._H}, ... )"

def save(self, filename : str) -> None:
    "si salva l'immagine dopo averla convertita in matrice di triple"
    images.save(self._asTriples(), filename)

def _asTriples(self) -> list[list[tuple[int,int,int]]] :
    "conversione della immagine da matrice di Color a matrice di triple"
    return [ [ pixel._asTriple() for pixel in riga ] for riga in self._img ]
↪]

def is_inside(self, x : float, y : float) -> bool:
    "verifico se le coordinate x,y sono dentro l'immagine"
    return 0 <= x < self._W and 0 <= y < self._H

def set_pixel(self, x : float, y : float, color) -> None:
    "cambio un pixel se è dentro l'immagine"
    x = round(x)
    y = round(y)
    if 0 <= x < self._W and 0 <= y < self._H:
        self._img[y][x] = color

def get_pixel(self, x : float, y : float) -> Colore:

```

```

    "leggo un pixel se è dentro l'immagine oppure torno None"
    x = max(0, min(round(x), self._W-1))
    y = max(0, min(round(y), self._H-1))
    return self._img[y][x]

def visualizza(self):
    "visualizzo l'immagine in Spyder"
    return images.visd(self._asTriples())

# fa comodo poter trovare i pixel intorno ad un punto, fino a distanza k
def vicini(self, x : int, y : int, k : int) -> list[Colore]:
    "torno i colori nei pixel 2k x 2k intorno al punto x,y"
    return [ self.get_pixel(X,Y)
             for X in range(x-k,x+k+1)
             for Y in range(y-k,y+k+1)
             if self.is_inside(X,Y)]

# copia?

```

4 Filtri come oggetti

Notate come la vita di un oggetto (istanza) sia fatta di DUE fasi - **creazione** con tutti i parametri e le info sue personali - **uso** con i suoi metodi

(in altri linguaggi dobbiamo anche “distruggere/disallocare” l’oggetto ma Python lo fa automaticamente)

Nell’uso dei filtri abbiamo dovuto usare trucchi come una lambda per passare parametri

Se li trasformiamo in oggetti i parametri li possiamo passare nella creazione e l’applicazione diventa più semplice

4.1 Definiamo il filtro generico e poi lo specializziamo aggiungendo funzionalità

- nell’`__init__` riceve tutti i parametri che gli serviranno
- ha un metodo `nuovo_pixel(self, pixel)` per i filtri che non dipendono dalla posizione
- ha un metodo `nuovo_pixel(self, immagine, x, y)` per i filtri che dipendono dalla posizione
- ha un metodo `reset(self)` per azzerarlo ed usarlo su altre immagini se memorizza informazioni quando applicato
- ha il metodo `applica(self, immagine)`

```

[5]: from pygraphviz import AGraph
G = AGraph(directed=True, rankdir='TD')
G.edge_attr['dir']='back'
G.node_attr['shape']='rect'
G.add_edges_from( [ ('GenericFilter', 'FiltroPixel'), ('GenericFilter',
↪'FiltroXY'), ('FiltroPixel', 'BiancoENero'),
                    ('FiltroPixel', 'Negativo'), ('FiltroPixel', 'Luminosità'), ('FiltroPixel',
↪'Contrasto'), ('FiltroPixel', 'RandomNoise'),

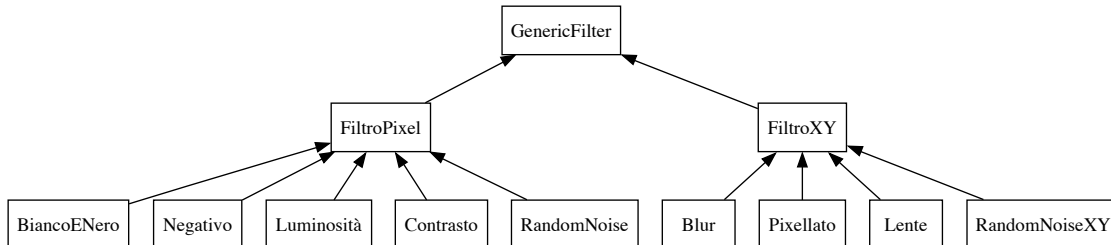
```

```

    ('FiltroXY', 'Blur'), ('FiltroXY', 'Pixellato'), ('FiltroXY', 'Lente'),
    ↪('FiltroXY', 'RandomNoiseXY'),
])
G.layout('dot')
G

```

[5]:



```

[6]: class GenericFilter: # tutti i filtri specializzano questa classe
    pass

```

```

[7]: class FiltroPixel (GenericFilter): # tutti i filtri indipendenti dalla
    ↪posizione
    "un filtro che NON dipende dalla posizione del pixel"
    # applicazione di un filtro per ottenere una nuova immagine
    def applica(self, immagine : Immagine) -> Immagine:
        "costruisco una nuova immagine con ciascun pixel trasformato tramite il
    ↪filtro"
        assert isinstance(immagine, Immagine), f"l'oggetto {immagine} non è una
    ↪Immagine"
        nuova_immagine = Immagine(larghezza=immagine._W, altezza=immagine._H)
        for y,riga in enumerate(immagine._img):
            for x,pixel in enumerate(riga):
                nuova_immagine._img[y][x] = self.nuovo_pixel(pixel) # applico
    ↪il filtro al pixel
        return nuova_immagine
    def nuovo_pixel(self, pixel : Colore) -> Colore : # deve
    ↪SEMPRE essere implementato dal filtro
        raise NotImplementedError()

```

```

[8]: class FiltroXY (GenericFilter): # tutti i filtri dipendenti dalla posizione
    "un filtro che DIPENDE dalla posizione del pixel"
    def applica(self, immagine : Immagine) -> Immagine:
        "applicazione di un filtro che conosce la posizione del pixel"
        assert isinstance(immagine, Immagine), f"{immagine} non è una Immagine"
        nuova_immagine = Immagine(larghezza=immagine._W, altezza=immagine._H)
        self.reset() # se il filtro contiene info da usi precedenti le azzero
        for y in range(immagine._H):

```

```

        for x in range(immagine._W):
            nuova_immagine._img[y][x] = self.nuovo_pixel_XY(immagine, x,y)
        return nuova_immagine
    def nuovo_pixel_XY(self, immagine : Immagine, x : int, y : int) -> Colore :
↳# deve SEMPRE essere implementato dal filtro XY
        raise NotImplementedError()
    def reset(self):                # per default non fa nulla (se il filtro non
↳memorizza nulla)
        pass

```

4.1.1 BiancoENero (FiltroPixel)

- genera pixel grigi della stessa luminosità dell'originale

```

[9]: class BiancoENero(FiltroPixel):
    def nuovo_pixel(self, pixel : Colore ) -> Colore :
        L = pixel.luminosità()
        return Colore(L,L,L)    # grigio di luminosità L

trecime = Immagine(filename='3cime.png')
BiancoENero().applica(trecime).visualizza()

```



4.1.2 Negativo (FiltroPixel)

- inverte la scala di luminosità

```

[10]: class Negativo(FiltroPixel):
    def nuovo_pixel(self, pixel : Colore ) -> Colore :
        return Colore.white - pixel

Negativo().applica(trecime).visualizza()

```



4.1.3 Cambio Luminosità (FiltroPixel)

- moltiplica il colore per un fattore k

```
[11]: class CambioLuminosità (FiltroPixel):  
    def __init__(self, k : float ) -> None:           # inizializzo col parametro k  
    ↪ *k* di luminosità  
        self._k = k  
    def nuovo_pixel(self, pixel : Colore ) -> Colore :  
        return pixel * self._k
```

```
CambioLuminosità(0.5).applica(trecime).visualizza()
```

```
CambioLuminosità(1.5).applica(trecime).visualizza()
```





4.1.4 Contrasto (FiltroPixel)

- avvicina/allontana i colori scuri e chiari al/dal grigio

```
[12]: class Contrasto(FiltroPixel):
    def __init__(self, k : float) -> None:           # inizializzo col parametro
    ↪ *k* di contrasto
        "Contrasto(k)"
        self._k = k
    def nuovo_pixel(self, pixel : Colore ) -> Colore :
        return Colore.grey + (pixel-Colore.grey)*self._k
```

```
Contrasto(0.8).applica(trecime).visualizza()
```

```
Contrasto(1.2).applica(trecime).visualizza()
```





4.1.5 Sfocatura/Blur (FiltroXY)

- dà al pixel il colore medio dei vicini fino a distanza k

```
[13]: class Blur(FiltroXY):
    def __init__(self, k : int):
        "inizializzo il filtro col parametro k"
        self._k = k
    def nuovo_pixel_XY(self, immagine : Immagine, x : int, y : int) -> Colore :
        "ciascun pixel è la media del gruppo grande 2k*2k che lo circonda"
        vicini = immagine.vicini(x, y, self._k )
        '''
        somma = Colore.black
        for v in vicini:
            somma += v
        return somma / len(vicini)
        '''
        return sum(vicini, Colore.black)/len(vicini)

Blur(2).applica(trecime).visualizza()
Blur(5).applica(trecime).visualizza()
```



4.1.6 Effetto pixellato (FiltroXY)

- per ogni pixel trova il quadretto che lo contiene
- calcola la **media dei pixel** nel quadretto
- **OTTIMIZZAZIONE** si ricorda questo valore (per non doverlo ricalcolare tante volte)
- da a tutti i pixel del quadretto lo stesso valore medio

```
[32]: class Pixellato(FiltroXY):
    "filtro che pixella l'immagine con la MEDIA dei pixel del quadretto"
    def __init__(self, size : int):
        "inizializzo il filtro con la dimensione del quadretto intera"
        self._size = size
        self._valori : dict[tuple[int,int], Colore ] = {}          # per ricordare
        ↪ i quadratini già calcolati
```

```

def reset(self) -> None :
    "per riusare lo stesso filtro su una diversa immagine lo devo resettare"
    self._valori = {}
def nuovo_pixel_XY(self, immagine : Immagine, x : int, y : int) -> Colore:
    "ciascun pixel è la media del gruppo grande 2k*2k che lo circonda"
    X = x - x % self._size + self._size//2    # centro del quadrato
    Y = y - y % self._size + self._size//2
    if not (X,Y) in self._valori:              # ottimizzazione che ricorda
↳ la media del quadretto in cui siamo
        vicini = immagine.vicini(X, Y, self._size//2 )
        self._valori[X,Y] = sum(vicini, Colore.black)/len(vicini)
    return self._valori[X,Y]

Pixellato( 5).applica(trecime).visualizza()    # applica resetta sempre il
↳ filtro
Pixellato(10).applica(trecime).visualizza()
Pixellato(15).applica(trecime).visualizza()

```





4.1.7 Effetto Lente (FiltroXY)

- all'esterno del cerchio della lente lascia l'immagine uguale
- all'interno legge i pixel che stanno ad una distanza dal centro della lente maggiorata/diminuita di un fattore k

```
[15]: from math import dist
class Lente(FiltroXY):
    def __init__(self, xc : int, yc : int,
                 raggio : int, ingrandimento : float) -> None:
        "inizializzo il filtro con posizione e raggio della lente e fattore di
        ↳ingrandimento"
        self._xc = xc
        self._yc = yc
```

```

self._raggio = raggio
self._ingrandimento = ingrandimento

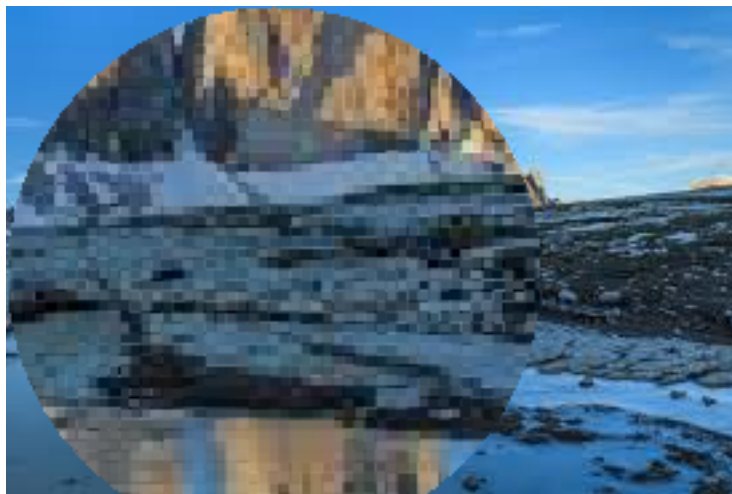
def nuovo_pixel_XY(self, immagine : Immagine, x : int, y : int):
    """ciascun pixel che sta dentro la lente
       è preso da quello che sta sulla retta dal centro della lente
       ad una distanza aumentata di ingrandimento
    """
    D = dist((x,y),(self._xc,self._yc))
    if D <= self._raggio:
        # cerca il pixel giusto
        x = round(self._xc + (x-self._xc) * self._ingrandimento)
        y = round(self._yc + (y-self._yc) * self._ingrandimento)
    return immagine.get_pixel(x,y)

```

```

Lente(100,100,100,0.5).applica(trecime).visualizza()
Lente(100,100,100,1.5).applica(trecime).visualizza()

```





4.1.8 Rumore sul pixel (FiltroPixel)

- RandomNoise: aggiunge al pixel una variazione di **colore** casuale da -k a +k per ogni canale
- RandomLight: aggiunge al pixel una variazione di **grigio** casuale da -k a +k uguale per tutti i canali

```
[16]: class RandomNoise(FiltroPixel):
    "filtro che aggiunge luminosità casuale separatamente a ciascun canale RGB"
    def __init__(self, k : int) -> None :
        self._k = k

    def nuovo_pixel(self, colore : Colore) -> Colore :
        return colore + Colore.random(-self._k, +self._k)

class RandomLight(RandomNoise):
    # Specializzo RandomNoise
    ↪cambiando solo il calcolo del pixel
    "filtro che aggiunge la stessa luminosità casuale ai tre canali RGB"
    def nuovo_pixel(self, colore : Colore) -> Colore :
        L = randint(-self._k, +self._k)
        return colore + Colore(L,L,L)

RandomNoise(50).applica(trecime).visualizza()
RandomLight(50).applica(trecime).visualizza()
```




4.1.9 Rumore sulla posizione del pixel (FiltroXY)

- sceglie un pixel vicino entro distanza k

```
[17]: from random import randint

class RandomNoiseXY(FiltroXY):
    def __init__(self, k : int) -> None :
        self._k = k

    def nuovo_pixel_XY(self, immagine : Immagine, x : int, y : int) -> Colore:
        X = x + randint(-self._k, self._k)
        Y = y + randint(-self._k, self._k)
        return immagine.get_pixel(X,Y)
```



```
RandomNoiseXY(5).applica(trecime).visualizza()
```



5 Grazie all' OOP

- **SEPARAZIONE DELLA CREAZIONE DALL'USO:** possiamo inserire nell'oggetto i suoi parametri personali senza usare **lambda**
 - esempio, **k** del contrasto, (**xc,yx, R e K**) della lente
 - ma quando li usiamo l'oggetto li conosce e non dobbiamo più fornirli
- **INTERFACCIA UGUALE:** anche se i due tipi di filtro sono diversi si applicano allo stesso modo!!! **applica(Immagine)**
 - nomi uguali dei metodi -> oggetti intercambiabili (ricordate il “Duck Typing”?)
 - l'applicazione che li **usa** non deve preoccuparsi di come sono fatti (tranne in questo caso per la loro creazione)
- **RIUSO DEL CODICE:** ciascun filtro eredita le funzionalità comuni dalla superclasse e la estende
 - una unica realizzazione delle funzionalità comuni -> meno errori di copy/paste e di aggiornamento

6 Altro esempio: Disegno di figure sulle immagini

- una Figura ha:
 - un colore
 - eventuali altri parametri
- e può:
 - essere disegnata su una immagine in una certa posizione x,y

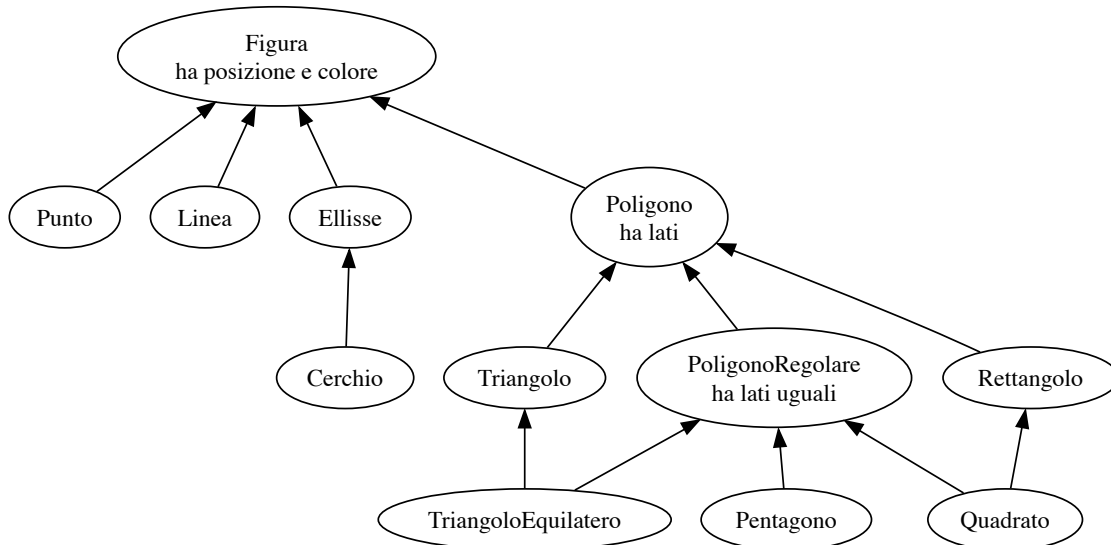
```
[18]: G = AGraph(directed=True)
G.edge_attr['dir']='back'
G.add_node('Figura',label="Figura\nha posizione e colore")
```

```

G.
↪add_nodes_from(['Punto','Linea','Ellisse','Cerchio','TriangoloEquilatero','Quadrato',
↪'Pentagono',' Rettangolo',' Triangolo'])
G.add_node('Poligono', label="Poligono\nha lati")
G.add_node('PoligonoRegolare', label="PoligonoRegolare\nha lati uguali")
G.add_edges_from([('Figura', 'Punto'),('Figura', 'Linea'),('PoligonoRegolare',
↪'Quadrato'), ('PoligonoRegolare', 'Pentagono'),
('Figura', 'Ellisse'),('Ellisse', 'Cerchio'), ('Poligono',
↪'PoligonoRegolare'), ('PoligonoRegolare', 'TriangoloEquilatero'),
('Figura', 'Poligono'), ('Poligono', 'Rettangolo'),
↪('Rettangolo', 'Quadrato'), ('Poligono', 'Triangolo'),
('Triangolo', 'TriangoloEquilatero')])
G.layout('dot')
G

```

[18]:



6.1 Figura e Punto

- tutte le Figure hanno un colore ed una posizione e possono essere disegnate
- il punto disegna un pixel di quel colore nella posizione

```

[19]: class Figura:
    def __init__(self, x: float, y: float, colore : Colore = Colore.black):
        self._colore = colore # ricordo il colore
        self._x = x # e la
        self._y = y # posizione

    def disegna(self, immagine : Immagine) -> None:
        raise NotImplementedError

```

```

class Punto (Figura):
    def disegna(self, immagine : Immagine) -> None:
        immagine.set_pixel(self._x, self._y, self._colore)

```

6.2 Linea (segmento)

- rappresentata da un punto iniziale (x,y) ed uno finale (x1, y1) e un colore

```

[20]: import math, random
class Linea (Figura):
    def __init__(self, x: float,y: float, x1: float, y1: float, colore:Colore):
        super().__init__(x, y, colore)          # eseguo l'inizializzazione
        ↪della superclasse
        self._x1 = x1                          # ed aggiungo come attributi
        ↪le coordinate
        self._y1 = y1                          # del secondo estremo

    def disegna(self, immagine : Immagine) -> None:
        x,x1 = self._x,self._x1
        y,y1 = self._y,self._y1
        dx = x1 - x
        dy = y1 - y
        if dx==0 and dy==0: return # evito divisioni per 0
        if abs(dx) > abs(dy):
            if dx<0:      x,y,x1,y1 = x1,y1,x,y   # scambio i due PUNTI se x>x1
            ↪(ATTENZIONE ORA E' CORRETTO)
            m = dy/dx
            for X in range(round(x), round(x1)+1):
                Y = m*(X-x) + y
                immagine.set_pixel(X,Y,self._colore)
        else:
            if dy<0:      x,y,x1,y1 = x1,y1,x,y   # scambio i due PUNTI se y>y1
            ↪(ATTENZIONE ORA E' CORRETTO)
            m = dx/dy
            for Y in range(round(y), round(y1)+1):
                XX = m * (Y-y) + x
                immagine.set_pixel(XX,Y,self._colore)

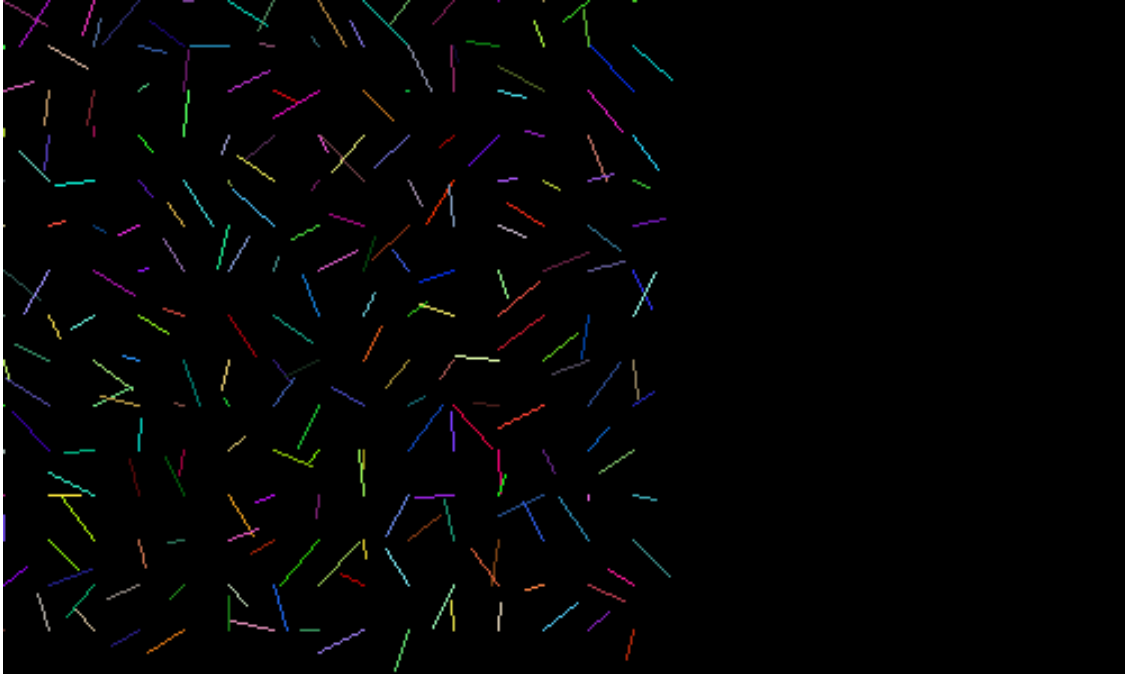
```

```

[21]: canvas = Immagine(500, 300)
for x in range(0,300,20):
    for y in range(0,300,20):
        dx = randint(-20,20)
        dy = randint(-20,20)
        Linea(x, y, x+dx, y+dy, Colore.random()).disegna(canvas)

canvas.visualizza()

```

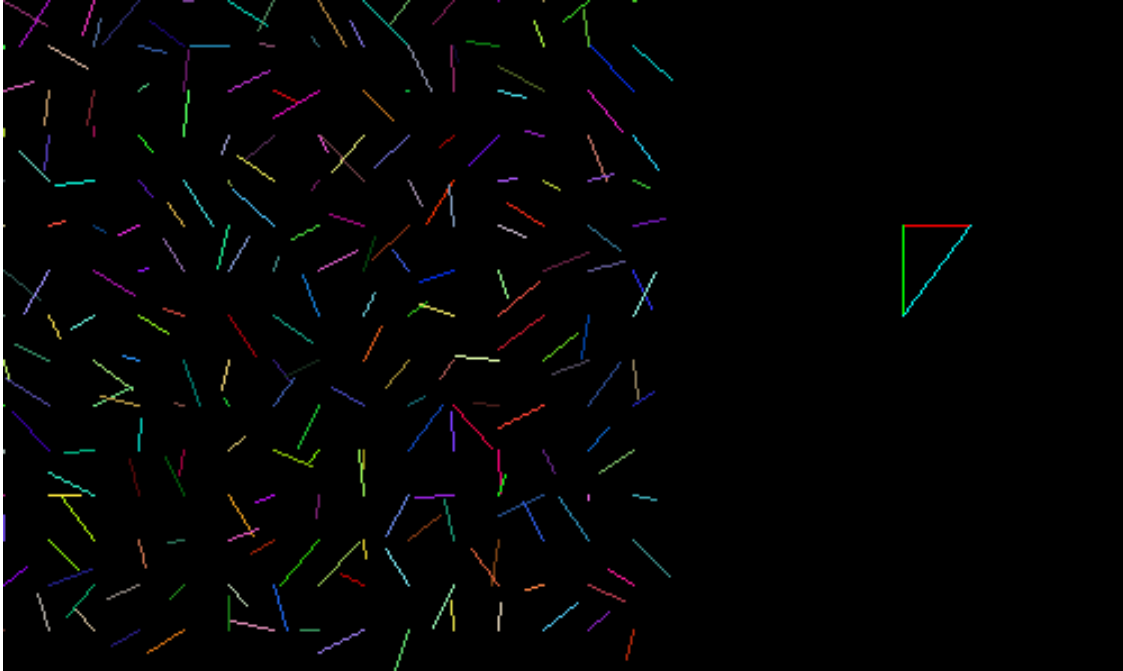


6.3 Poligono

- un qualsiasi poligono è fatto da un gruppo di linee
- (non verifichiamo che siano consecutive o che sia una superficie chiusa)

```
[22]: class Poligono(Figura):
    "Un poligono è una figura fatta di linee"
    def __init__(self, linee : list[Linea]):
        self._linee = linee
    def disegna(self, immagine: Immagine): # le disegnamo una ad una
        for l in self._linee:
            l.disegna(immagine)

Poligono([
    Linea(400, 100, 430, 100, Colore.red),
    Linea(400, 100, 400, 140, Colore.green),
    Linea(430, 100, 400, 140, Colore.cyan),
]).disegna(canvas)
canvas.visualizza()
```

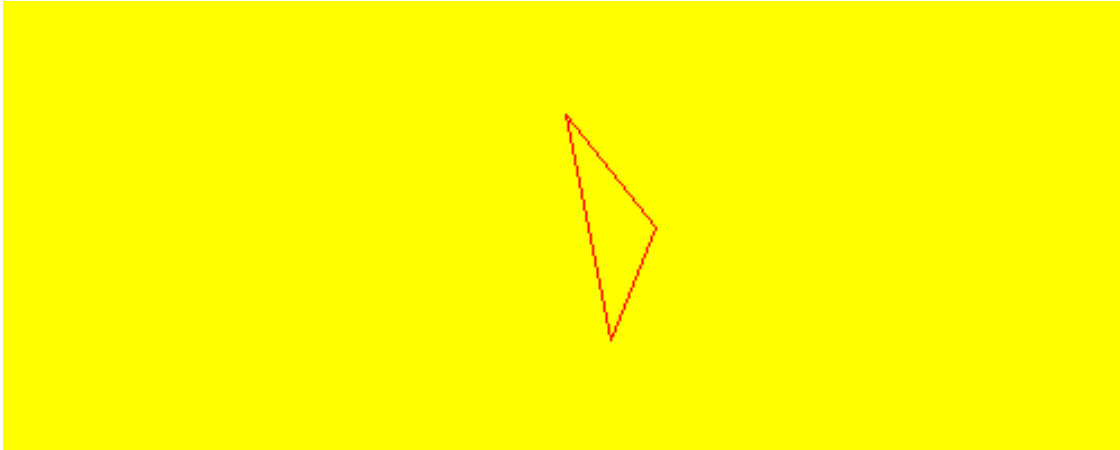


6.4 Triangolo

- dati 3 punti, ne costruisce le tre linee

```
[23]: class Triangolo(Poligono):  
    "Un Triangolo è un poligono con 3 lati che connettono 3 punti"  
    def __init__(self, x : float, y : float,  
                  x1 : float, y1 : float,  
                  x2 : float, y2 : float, colore : Colore):  
        lato1 = Linea(x, y, x1,y1, colore)  
        lato2 = Linea(x1,y1,x2,y2, colore)  
        lato3 = Linea(x2,y2,x, y, colore)  
        super().__init__([lato1, lato2, lato3]) # delego alla superclasse  
    ↪ Poligono di ricordare le 3 linee e disegnarle
```

```
[24]: canvas = Immagine(500, 200, Colore.yellow)  
Triangolo(250,50, 290, 100, 270, 150, Colore.red).disegna(canvas)  
canvas.visualizza()
```



6.5 Rettangolo

- dato l'angolo sopra a sinistra, larghezza, altezza e inclinazione
- costruisce le 4 linee

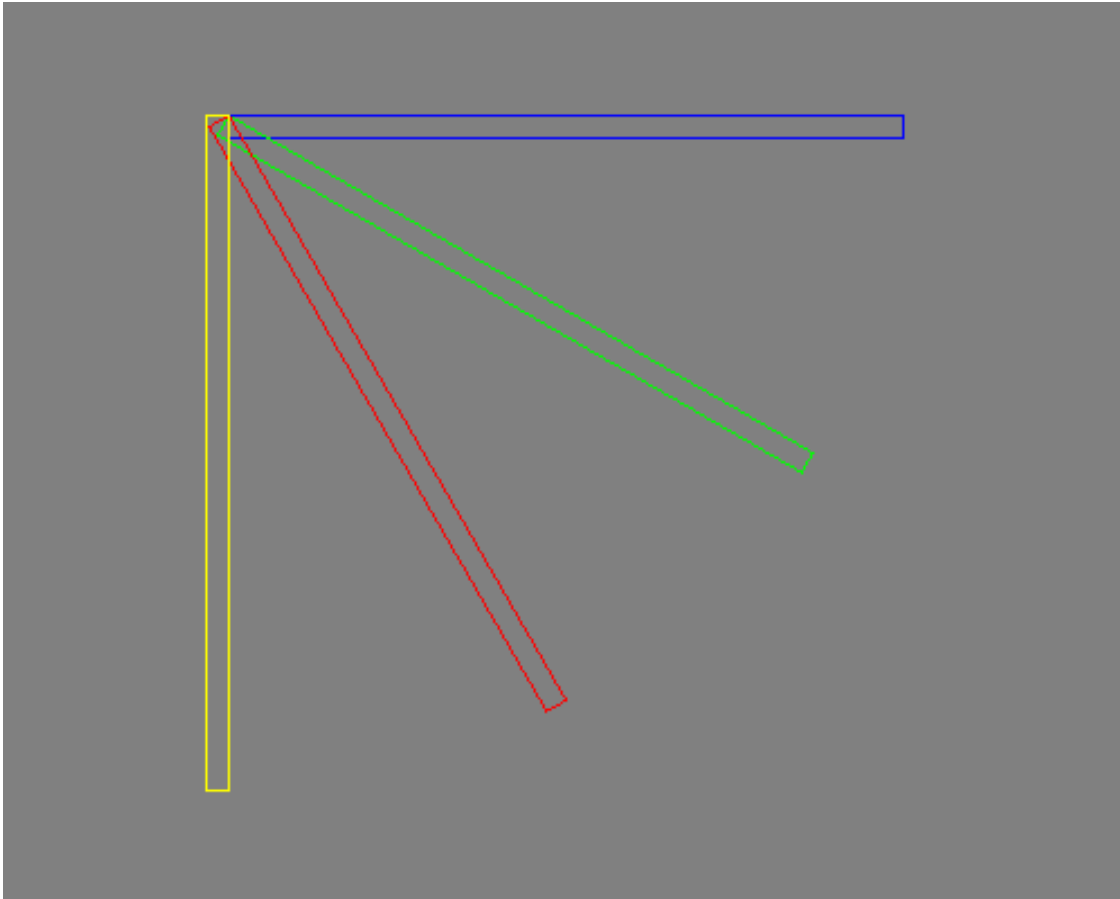
```
[25]: class Rettangolo(Poligono):
    "Un Rettangolo è un poligono con 4 lati perpendicolari"
    def __init__(self, x : float, y : float, larghezza : float, altezza : float,
    ↪angolo : float, colore : Colore ):
        self._larghezza = larghezza
        self._altezza = altezza
        self._angolo = angolo
        orizzontale = math.radians(angolo)           # direzione del lato
    ↪superiore in radianti
        verticale = math.radians(angolo+90)         # direzione del lato
    ↪sinistro in radianti
        # calcolo le posizioni dei 4 vertici
        x1 = x + math.cos(orizzontale)*larghezza    # vertice in alto a destra
        y1 = y + math.sin(orizzontale)*larghezza
        x2 = x1 + math.cos(verticale)*altezza       # vertice in basso a destra
        y2 = y1 + math.sin(verticale)*altezza
        x3 = x + math.cos(verticale)*altezza       # vertice in basso a sinistra
        y3 = y + math.sin(verticale)*altezza
        sopra = Linea(x, y, x1, y1, colore)
        sinistra = Linea(x, y, x3, y3, colore)
        sotto = Linea(x3, y3, x2, y2, colore)
        destra = Linea(x1, y1, x2, y2, colore)
        super().__init__([sopra, sotto, sinistra, destra])
```

```
[35]: canvas = Immagine(500, 400, Colore.grey)
    Rettangolo(100, 50, 300, 10, 0, Colore.blue ).disegna(canvas)
```

```

Rettangolo(100, 50, 300, 10, 30, Colore.green ).disegna(canvas)
Rettangolo(100, 50, 300, 10, 60, Colore.red   ).disegna(canvas)
Rettangolo(100, 50, 300, 10, 90, Colore.yellow).disegna(canvas)
canvas.visualizza()

```



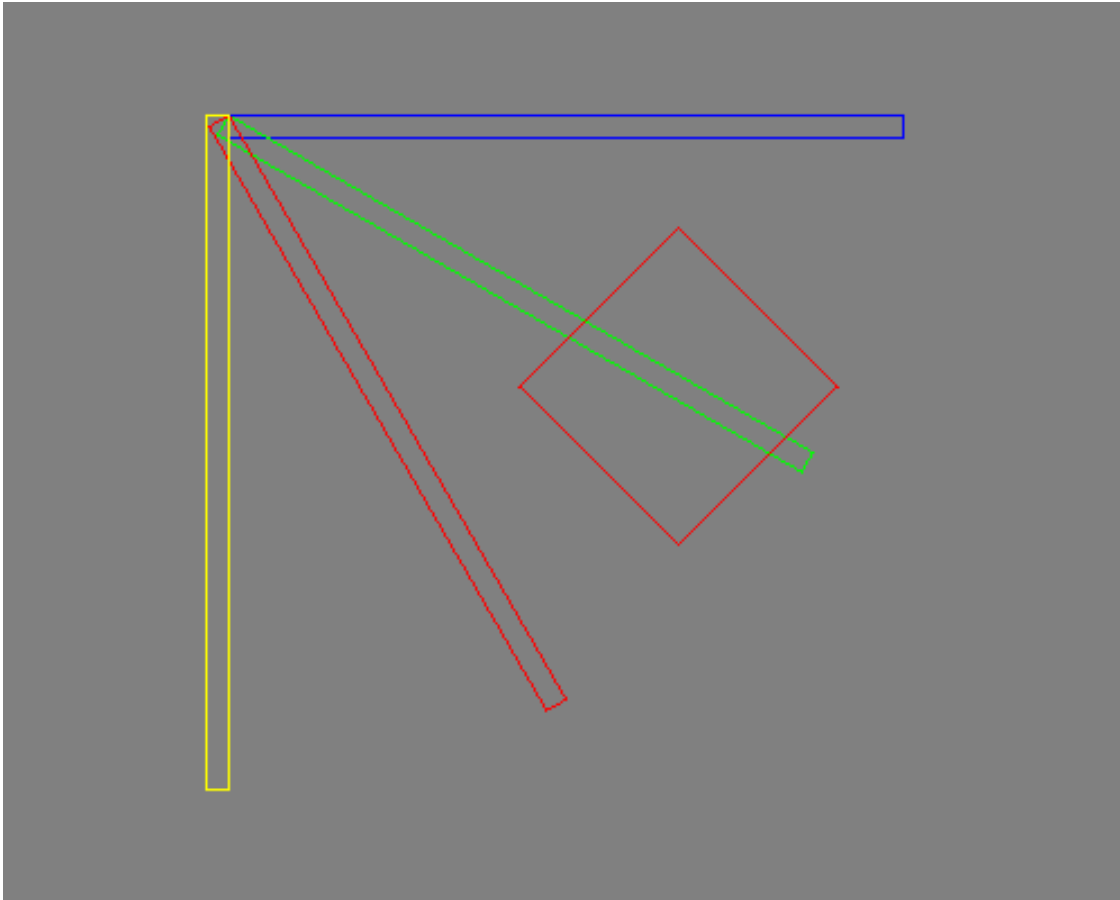
6.6 un Quadrato è un rettangolo con lati uguali

```

[36]: class Quadrato(Rettangolo):
        def __init__(self, x: float, y: float, lato : int, direzione : float,
        ↪ colore : Colore):
            "un Quadrato è un Rettangolo con lati uguali"
            super().__init__(x, y, lato, lato, direzione, colore) # delego al
            ↪ rettangolo l'inizializzazione

Quadrato(300, 100, 100, 45, Colore.red).disegna(canvas)
canvas.visualizza()

```



6.7 Un poligono regolare con N lati

- dato il centro e il raggio del cerchio in cui è iscritto
- ci calcoliamo i vertici e quindi tutti i lati

```
[33]: class PoligonoRegolare(Poligono):
    def __init__(self, xc: float, yc: float, R: float, N:int, Alfa:float,
    ↪colore:Colore):
        "Creo un poligono regolare con N lati, centro xc,yc, raggio R e
    ↪rotazione Alfa (in gradi)"
        vertici = []
        for direzione in range(N):                                # per ciascuno di N
    ↪vertici
            radianti = math.radians(direzione*360/N+Alfa)        # calcolo l'angolo
    ↪in radianti
            X = xc + math.cos(radianti)*R                        # e le sue
    ↪coordinate usando cos e sin
            Y = yc + math.sin(radianti)*R
            vertici.append((X,Y))
```



```

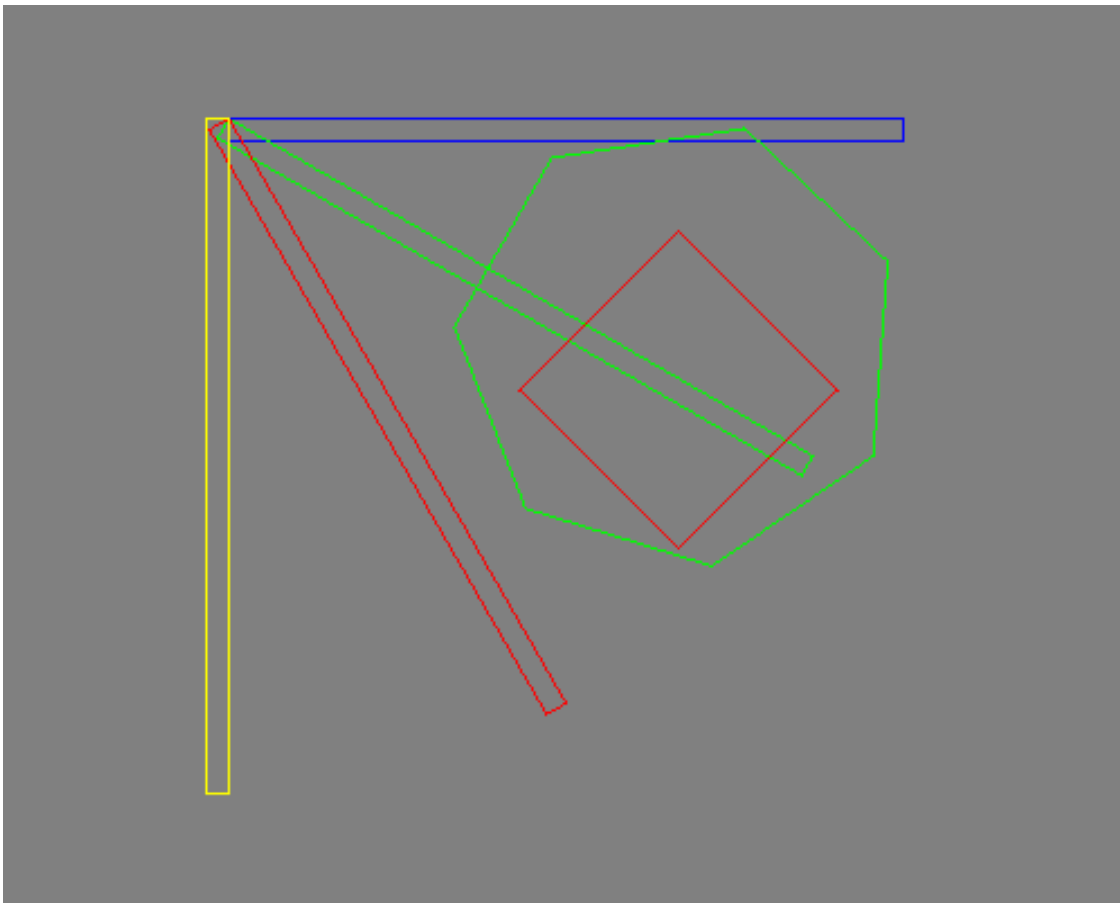
lati = []
for i in range(N):
    lati.append(Linea(*vertici[i],*vertici[i-1],colore)) # creo una
↳ linea tra il punto corrente ed il precedente
    # NOTA: il precedente dell'indice 0 è l'indice -1 che torna l'ultimo
↳ punto e chiude il poligono
    super().__init__(lati) # delego al
↳ Poligono ricordarsi le linee

```

```

[37]: PoligonoRegolare(300, 150, 100, 7, 30, Colore.green).disegna(canvas)
      canvas.visualizza()

```



```

[38]: class TriangoloEquilatero(PoligonoRegolare):
      def __init__(self, xc: float, yc: float, R : float, Alpha: float, colore:
↳ Colore):
          "un triangolo equilatero è un poligono regolare con 3 lati"
          super().__init__(xc, yc, R, 3, Alpha, colore)

```

```
[39]: TriangoloEquilatero(300, 150, 100, 30, Colore.cyan).disegna(canvas)
      canvas.visualizza()
```

