:::{list-table} :header-rows: 1 :align: center :widths: 90 10

- ▪ Fondamenti di programmazione

  Canale A-L - Prof. Sterbini

  AA 25-26 - Lezione 15

  - :::{image}attachment:5a837f06-202c-4c6e-9259-dd7a6dc01c97.png :width: 200px ::: :::

RECAP: CLASSI e oggetti

- **classi**: hanno **attributi** (di classe) e **metodi** comuni a tutte le istanze (individui) della classe
- **istanze**: hanno **attributi personali** e rispondono ai metodi della loro classe

Ereditarietà tra classi

Le classi che definiscono i tipi di oggetti possono **ereditare** degli attributi e dei metodi dalle **superclassi**

- si dice che la sottoclasse **ESTENDE** la classe da cui eredita
  - cioè le può aggiungere nuovi attributi e nuovi metodi
  - può sostituire i metodi con nuove realizzazioni
- si evita di riscrivere codice comune a più tipi di oggetti
- è facile **specializzare** il comportamento di sottoclassi di oggetti
- quando una istanza vuole eseguire un metodo o usare un attributo la ricerca avviene **dal basso verso l'alto**
  - quindi viene eseguita la versione **più specializzata** del metodo o dell'attributo

In [1…

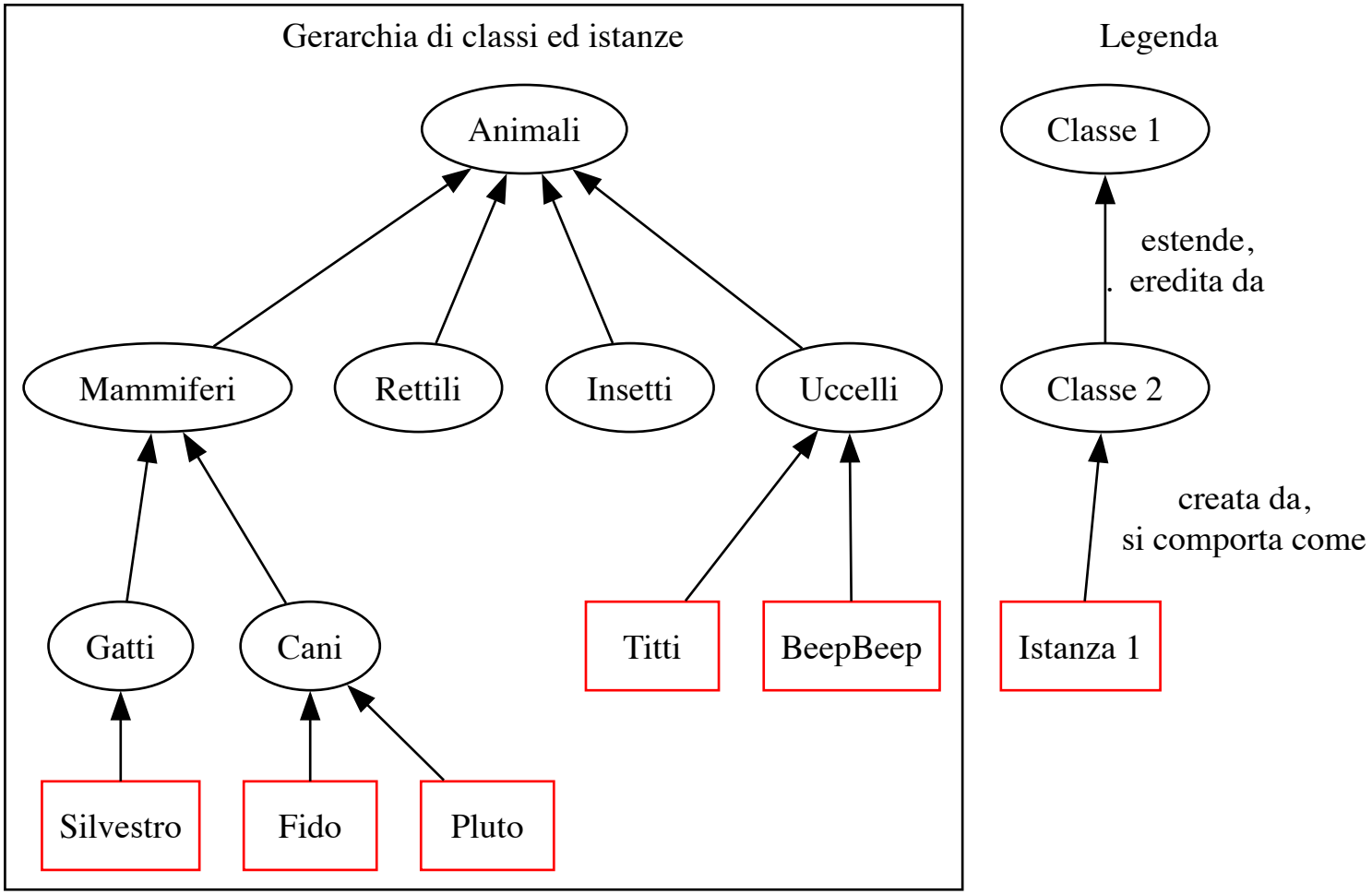
```
from pygraphviz import AGraph
```

```
G = AGraph(directed=True, rankdir='TD')
G.edge_attr['dir'] = 'back'
G.add_nodes_from(['Titti','Silvestro','Fido','Pluto','BeepBeep','Istanza 1'],shape='rect',color='red')
G.add_nodes_from(['Animali','Mammiferi','Gatti','Cani','Rettili','Insetti','Uccelli','Classe 1','Classe
G.add_edges_from([('Animali','Mammiferi'),('Animali','Rettili'),('Animali','Insetti'),('Animali','Uccel
                  ('Mammiferi','Gatti'),('Mammiferi','Cani')])
G.add_edges_from([('Gatti','Silvestro'),('Cani','Fido'),('Cani','Pluto'),('Uccelli','Titti'),
                  ('Uccelli','BeepBeep')])
G.add_edge('Classe 1','Classe 2',label='   estende,\n.  eredita da')
G.add_edge('Classe 2','Istanza 1',label='   creata da,\n   si comporta come')
G.subgraph(['Animali','Rettili','Insetti','Mammiferi','Uccelli','Gatti','Cani',
            'Titti','Silvestro','Fido','Pluto','BeepBeep'],
          label='Gerarchia di classi ed istanze', name='cluster_1')
G.subgraph(['Classe 1','Classe 2','Istanza 1'], label='Legenda', color='invis', name='cluster_2')
G.layout('dot')
```

In [2…

```
## ESEMPIO di GERARCHIA di classi
G
```

Out [2…



Colori

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

In [3…

```
import images
from random import randint

class Colore:
```

```
    white : 'Colore' # indichiamo che gli attributi di classe hanno tipo 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(min,max)
    @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             # media
        return 0.299*self._R + 0.587*self._G + 0.114*self._B # media pesata

    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"
        return f"Colore({self._R}, {self._G}, {self._B})"
```

In [4…

```
# solo dopo aver definito Colore posso aggiungere attributi di classe 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
```

Out[4… `Colore(127.5, 127.5, 0.0)`

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**

In [5…

```
import images
import math

class Immagine:

    def __init__(self, larghezza:int, altezza:int, sfondo:Colore=Colore.black):
        # FIXME: dovrei controllare gli argomenti
        self._W = larghezza
        self._H = altezza
        self._img  = [ [sfondo for _ in range(larghezza) ] for _ in range(altezza)  ]

    @classmethod
    def load(cls, filename : str) -> 'Immagine':
        # FIXME: dovrei controllare gli argomenti
        img = images.load(filename)
        W = len(img[0])
        H = len(img)
        myself = cls(W,H)
        # letta la immagine la converto in una matrice di Colore
```

```
        myself._img = [ [ Colore(r,g,b) for r,g,b in riga  ] for riga in img  ]
        return myself

    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 self.is_inside(x,y):
            self._img[y][x] = color

    def get_pixel(self, x : float, y : float) -> Colore:
        "leggo un pixel se è dentro l'immagine oppure torno il pixel sul bordo, accetto coordinate float
        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?
```

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 automaticamante)

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

Definiamo il filtro generico e poi lo specializziamo aggiungendo funzionalità

- nell' `__init__` riceve tutti i parametri che gli serviranno quando viene applicato
- 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)** che genera la nuova immagine applicandolo a tutti i pixel

Grazie all' OOP (Object Oriented Programming)

- **SEPARAZIONE DELLA CREAZIONE DALL'USO**: possiamo inserire nell'oggetto i suoi parametri personali senza usare **lambda**
  - esempio, **k** del contrasto, **(xc,yc, R e K)** della lente

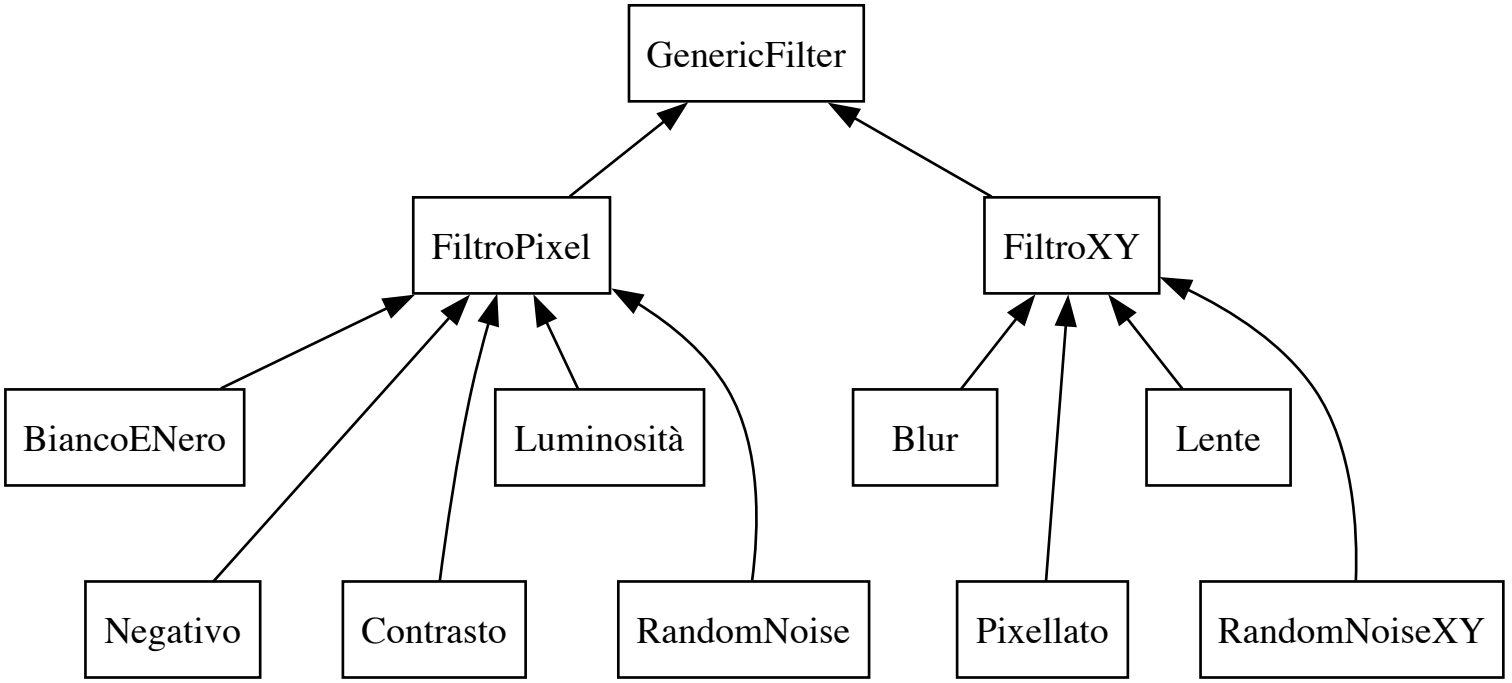
    - ma quando li usiamo l'oggetto li conosce e non dobbiamo più fornirli
- **INTERFACCIA UGUALE**: anche se i due tipi di filtro sono diversi si applicano allo stesso modo!!! **applica(Immagine)**
    - nomi uguali dei metodi -> oggetti intercambiabili
        - "Duck Typing": se un oggetto fa "Quack" come una papera è una papera
    - 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

In [6…
```
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' , 'B
    ('FiltroPixel', 'Negativo'), ('FiltroPixel', 'Luminosità'), ('FiltroPixel', 'Contrasto'), ('FiltroPi
    ('FiltroXY', 'Blur'), ('FiltroXY', 'Pixellato'), ('FiltroXY', 'Lente'), ('FiltroXY', 'RandomNoiseXY
G.add_edges_from( [ ('BiancoENero', 'Negativo'), ('Luminosità', 'Contrasto'), ('Luminosità', 'RandomNois
    ('Blur', 'Pixellato'), ('Lente', 'RandomNoiseXY'),],color='invis')
G.layout('dot')
```

In [7…
```
G
```

Out [7…



In [8…

```python
class GenericFilter:  # tutti i filtri specializzano questa classe
    pass              # necessario per la sintassi
```

FiltroPixel

- tutte le sottoclassi ereditano il metodo **`applica(Colore)->Colore`** che trasforma ciascun pixel
- non conosce la posizione del pixel
- ogni sottoclasse deve solo calcolare il nuovo colore con **`nuovo_pixel(Colore)->Colore`**

In [9…

```python
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

    # definisco un metodo che dà errore, così se nelle sottoclassi mi dimentico di crearlo me ne accorg
    def nuovo_pixel(self, pixel : Colore) -> Colore :       # deve SEMPRE essere implementato dal filtr
        "metodo che calcola il valore del nuovo pixel"
        raise NotImplementedError()
```

BiancoENero (FiltroPixel)

- genera pixel grigi della stessa luminosità dell'originale

In [1…

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

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

Negativo (FiltroPixel)

- inverte la scala di luminosità

In [1…

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

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

Cambio Luminosità (FiltroPixel)

- moltiplica il colore per un fattore k

NOTA i parametri del filtro li inserisco alla **CREAZIONE** dell'oggetto e lui li conosce ed applica come serve

- non dobbiamo più usare trucchi (lambda)

In [1…

```
class CambioLuminosità (FiltroPixel):
    def __init__(self, k : float ) -> None:          # inizializzo col parametro *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()
```

Contrasto (FiltroPixel)

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

In [1…

```
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()
```

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

In [1…

```
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 pix
    "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(100).applica(trecime).visualizza()
RandomLight(100).applica(trecime).visualizza()
```

FiltroXY

- tutte le sottoclassi ereditano il metodo **`applica(Immagine)->Immagine`** che trasforma tutti i pixel
- conosce la posizione del pixel da trasformare
- potrebbe ricordare delle info
  - viene resettato prima di applicare la trasformazione alla immagine

Ogni sottoclasse implementa

- **`nuovo_pixel_XY(Immagine,X,Y)->Colore`** (obbligatorio) che trasforma il pixel
- **`reset()`** (opzionale) che azzera le info che il filtro potrebbe aver memorizzato (per ottimizzazione)

In [1…

```
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 ricorda informazioni 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

    # deve SEMPRE essere implementato dal filtro XY
    def nuovo_pixel_XY(self, immagine : Immagine, x : int, y : int) -> Colore :
        "metodo che calcola il valore del nuovo pixel"
        raise NotImplementedError()

    # opzionale, va implementato solo se il filtro memorizza informazioni mentre lavora sulla immagine
    def reset(self):              # per default non fa nulla (se il filtro non memorizza nulla)
        "Se il filtro memorizza qualcosa può azzerarne il contenuto prima di eseguire applica"
        pass
```

Sfocatura/Blur (FiltroXY)

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

In [1…

```
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 )
        return sum(vicini, Colore.black)/len(vicini)
        ''' # oppure
        somma = Colore.black
        for v in vicini:
            somma += v
        return somma / len(vicini)
```

```
        '''

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

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

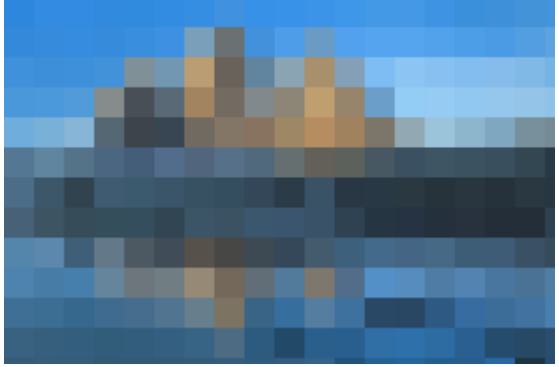
In [1…

```
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 calcolat
```

```
    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:             # se è la prima volta che sono in questo quadretto
            # 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()
```



Effetto Lente (FiltroXY)

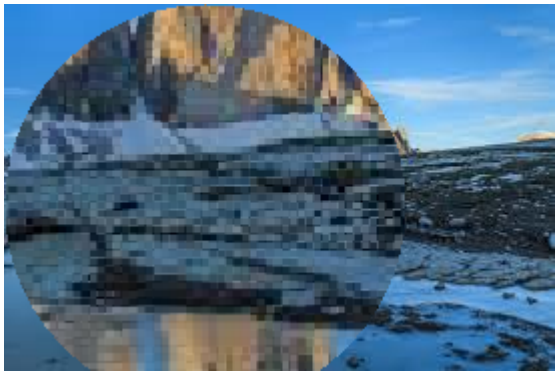
- all'esterno del cerchio della lente lascia l'immagine uguale
- all'interno legge i pixel che stanno ad una distanza dal centro della lente maggiorata/diminuita di un fattore k

In [1…

```
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()
```



Rumore sulla posizione del pixel (FiltroXY)

- sceglie un pixel vicino entro distanza k

In [1…

```
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()
```

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

In [2…

```
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('Spezzata', label="Spezzata\nha lati consecutivi")
G.add_node('Poligono', label="Poligono\nè chiuso")
G.add_node('PoligonoRegolare', label="PoligonoRegolare\nha lati uguali")
G.add_edges_from([('Figura', 'Punto'),('Figura', 'Linea'),('Figura','Spezzata'),('Figura', 'Ellisse'),
                  ('Spezzata', 'Poligono'),
                  ('PoligonoRegolare', 'Quadrato'),
                  ('PoligonoRegolare', 'Pentagono'),
    ('Ellisse', 'Cerchio'),
                  ('Poligono', 'PoligonoRegolare') , ('PoligonoRegolare' ,'TriangoloEquilatero'),
    ('Poligono','Rettangolo'), ('Rettangolo','Quadrato'), ('Poligono', 'Triangolo'),
    ('Triangolo','TriangoloEquilatero')])
```

```
G.layout('dot')
```

In [2…

```
G
```

Out[2…

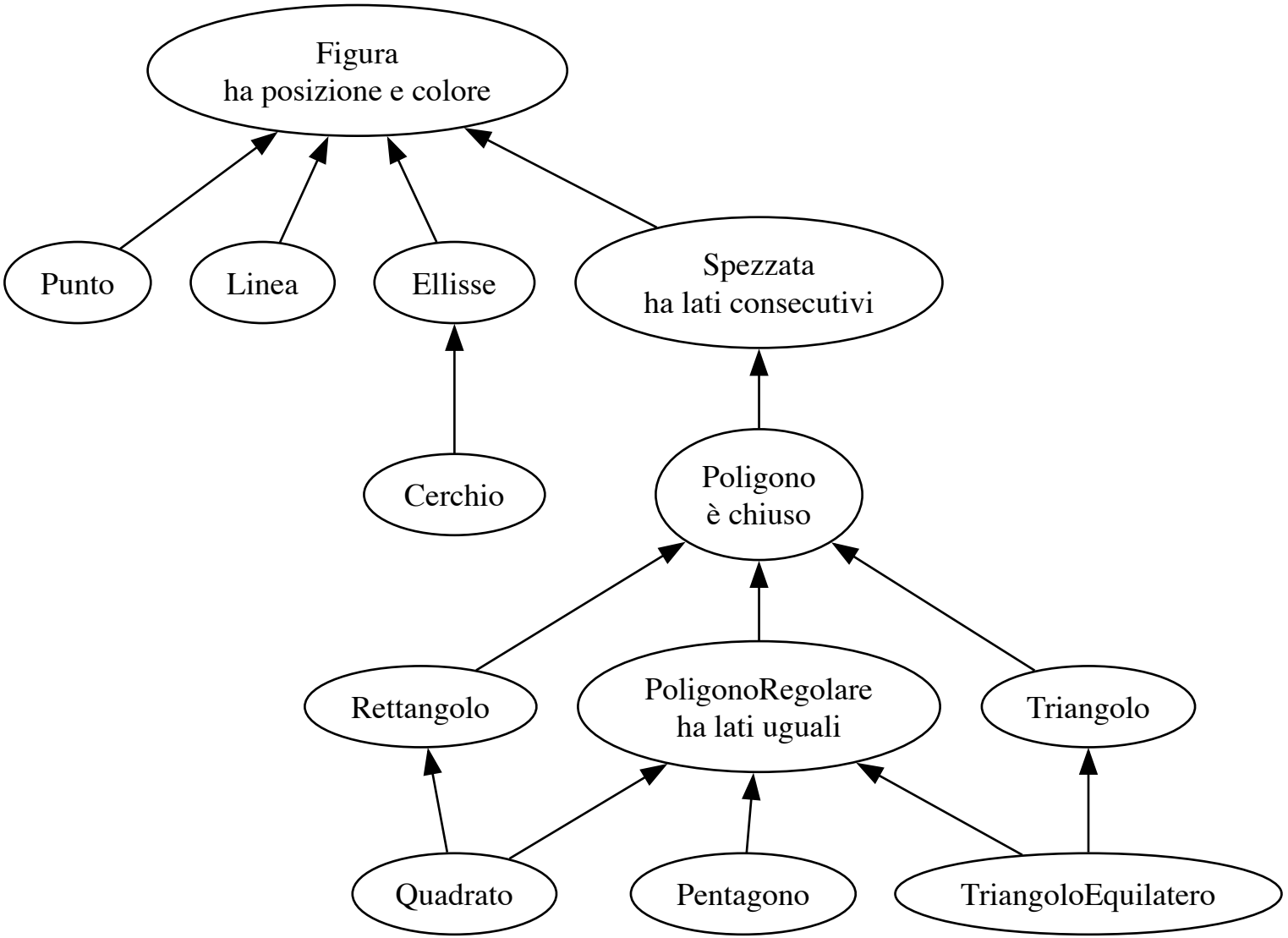


Figura e Punto

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

In [2…

```
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:       # DEVE essere implementato nelle sottoclassi
        raise   NotImplementedError

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

Linea (estende Figura)

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

In [2…

```
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 Figura
        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: # evito divisioni per 0
            immagine.set_pixel(x,y,self._colore)
            return
        if abs(dx) > abs(dy):
            if dx<0:
                x,y,x1,y1 = x1,y1,x,y   # scambio i due PUNTI se x>x1
            m = dy/dx
```

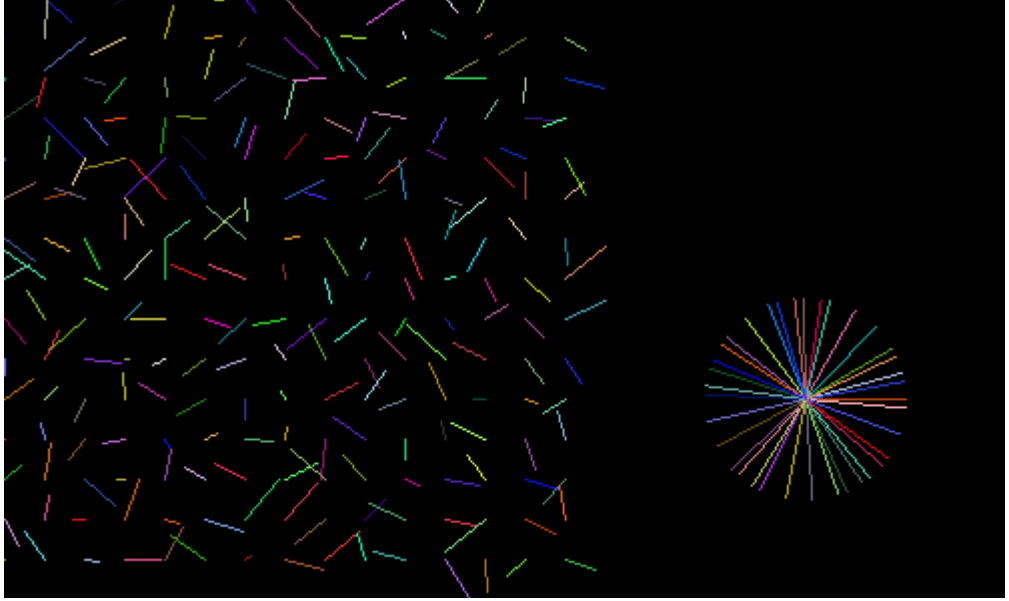
```
            for X in range(round(x), round(x1)+1):
                Y  = m*(X-x) + y
                immagine.set_pixel(X,Y,self._colore)
        else:
            if dy<0:
                x,y,x1,y1 = x1,y1,x,y   # scambio i due PUNTI se x>x1
            m = dx/dy
            for Y in range(round(y), round(y1)+1):
                XX  = m * (Y-y) + x
                immagine.set_pixel(XX,Y,self._colore)
```

In [2…

```
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)

from math import sin, cos
for x in range(0,360,10):
    dx = 50*cos(x)
    dy = 50*sin(x)
    Linea(400, 200, 400+dx, 200+dy, Colore.random()).disegna(canvas)

canvas.visualizza()
```



Spezzata (estende Figura)

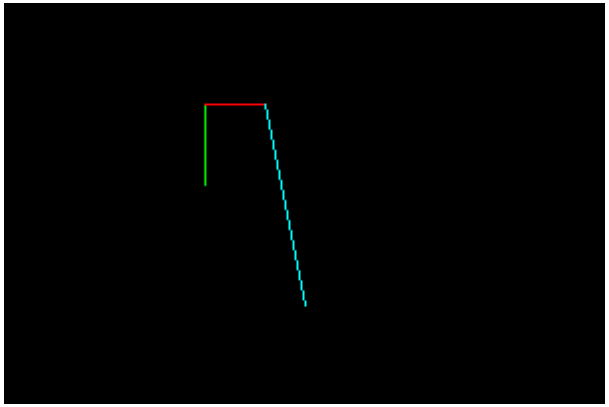
- una linea Spezzata è fatta da un gruppo di linee
- verifichiamo che siano consecutive

In [2…

```
class Spezzata(Figura):
    "Una spezzata è una figura fatta di linee consecutive"
    def __init__(self, linee : list[Linea]):
        for i in range(len(linee)-1):
            assert (linee[i]._x1,linee[i]._y1) == (linee[i+1]._x,linee[i+1]._y), "Segmenti non consecut
        self._linee  = linee
    def disegna(self, immagine: Immagine):   # le disegnamo una ad una
        for l in self._linee:
            l.disegna(immagine)

canvas = Immagine(300, 200)
Spezzata([
    Linea(100, 90, 100, 50, Colore.green),
    Linea(100, 50, 130, 50, Colore.red),
    Linea(130, 50, 150, 150, Colore.cyan),
]).disegna(canvas)
```

```
canvas.visualizza()
```



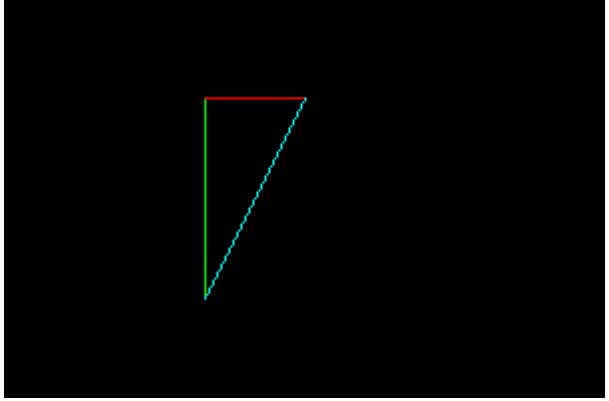
Poligono (estende Spezzata)

- un qualsiasi poligono è fatto da un gruppo di linee CHIUSO
- verifichiamo che siano consecutive e che sia chiuso

In [2…

```
class Poligono(Spezzata):
    "Un poligono è una Spezzata chiusa"
    def __init__(self, linee : list[Linea]):
        # delego alla superclasse Spezzata di controllare che siano consecutive e ricordarle
        super().__init__(linee)
        # controllo che siano chiuse
        assert (linee[-1]._x1,linee[-1]._y1) == (linee[0]._x,linee[0]._y), "Poligono non chiuso"

canvas = Immagine(300, 200)
Poligono([
    Linea(100, 150, 100,  50, Colore.green),
    Linea(100,  50, 150,  50, Colore.red),
    Linea(150,  50, 100, 150, Colore.cyan),
]).disegna(canvas)
canvas.visualizza()
```



Triangolo (estende Poligono)

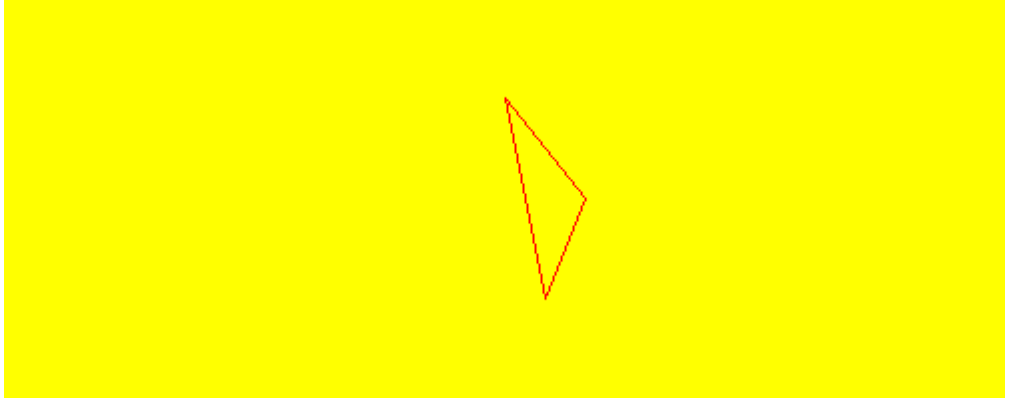
- dati 3 punti, ne costruisce le tre linee

In [2…

```
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)
        # delego alla superclasse Poligono di ricordare le 3 linee e disegnarle
        super().__init__([lato1, lato2, lato3])
```

In [2…

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



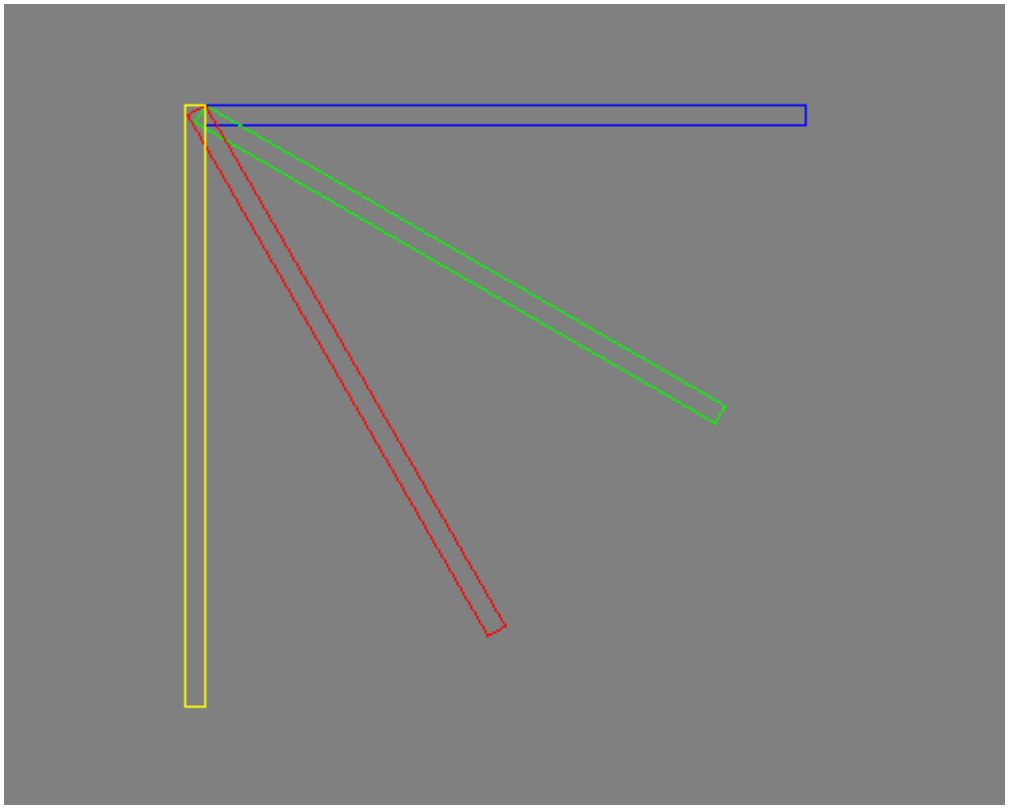
Rettangolo (estende Poligono)

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

In [2…

```
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
        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)
        destra     = Linea(x1, y1, x2, y2, colore)
        sotto      = Linea(x2, y2, x3, y3, colore)
        sinistra   = Linea(x3, y3, x,  y,  colore)
        super().__init__([sopra, destra, sotto, sinistra])
```

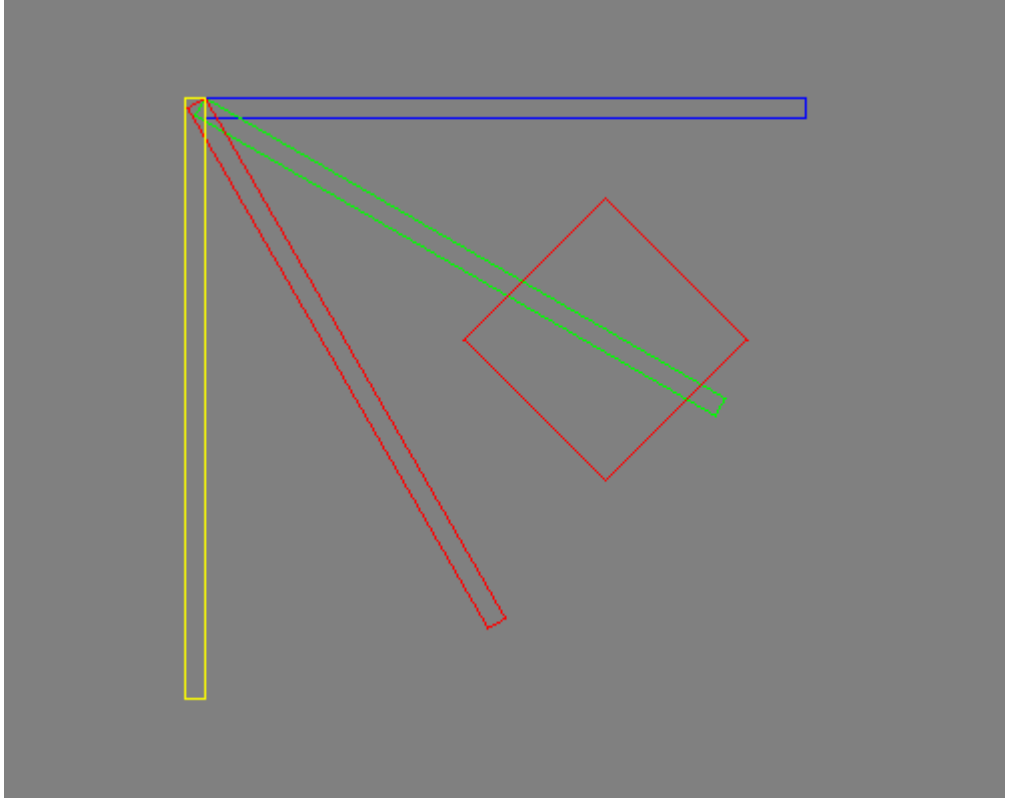
In [3…
```
canvas = Immagine(500, 400, Colore.grey)
Rettangolo(100, 50, 300, 10,  0, Colore.blue  ).disegna(canvas)
Rettangolo(100, 50, 300, 10, 30, Colore.green ).disegna(canvas)
Rettangolo(100, 50, 300, 10, 60, Colore.red   ).disegna(canvas)
Rettangolo(100, 50, 300, 10, 90, Colore.yellow).disegna(canvas)
canvas.visualizza()
```



un Quadrato è un rettangolo con lati uguali

In [3…
```
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()
```



Un poligono regolare con N lati

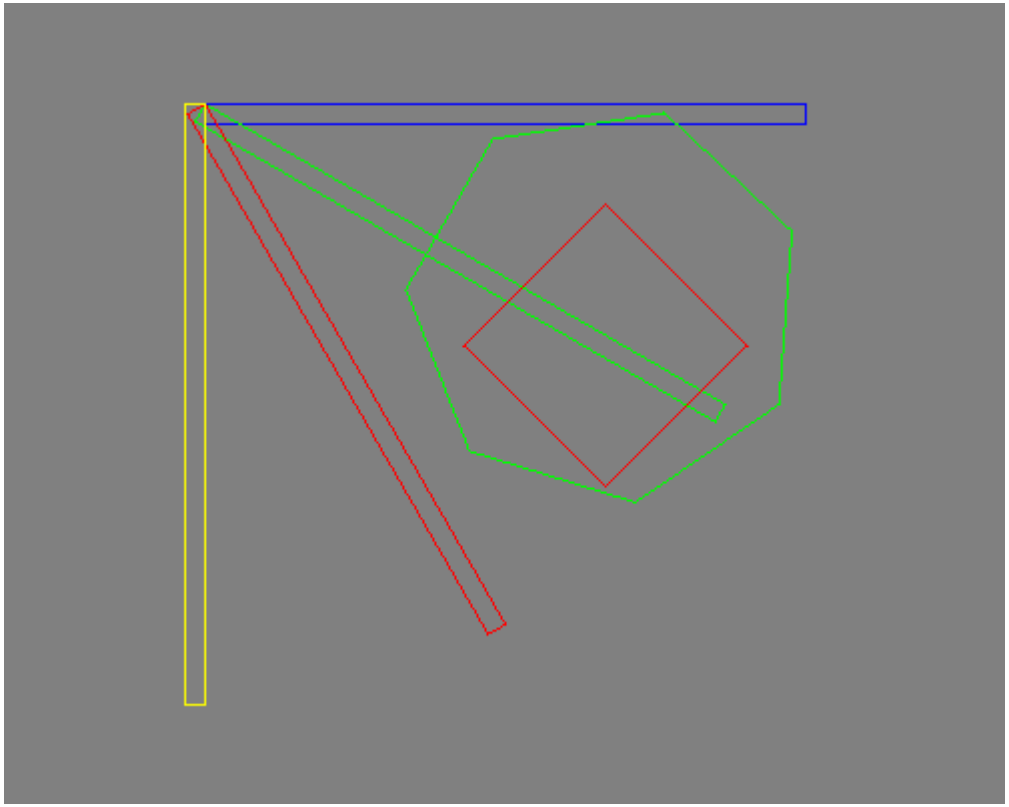
- dato il centro e il raggio del cerchio in cui è iscritto
- ci calcoliamo i vertici e quindi tutti i lati

In [3…

```python
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):                                  # per N volte
```

```
            # creo una linea tra il punto precedente e quello corrente
            # NOTA: il precedente del'indice 0 è l'indice -1 che torna l'ultimo punto e chiude il poligo
            lati.append(Linea(*vertici[i-1],*vertici[i],colore))
        super().__init__(lati)                              # delego al Poligono ricordarsi le linee
```

In [3…
```
PoligonoRegolare(300, 150, 100, 7, 30, Colore.green).disegna(canvas)
canvas.visualizza()
```



un TriangoloEquilatero è un PoligonoRegolare

In [3…
```
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)  # delego al PoligonoRegolare di creare le 3 linee
```

In [3…
```
TriangoloEquilatero(300, 150, 100, 50, Colore.cyan).disegna(canvas)
canvas.visualizza()
```