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

- ▪ Fondamenti di programmazione

  Canale A-L - Prof. Sterbini

  AA 25-26 - Lezione 14

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

RECAP: Immagini

- ritaglio/crop
- cut-paste
- filtri che **NON dipendono** dalla posizione
- filtri che **dipendono** dalla posizione

OGGI

- Effetto swirl/vortice e onda
- Programmazione ad oggetti
  - immagini come oggetti
  - colori come oggetti e matematica dei colori

In [1…
```
# estensione che usa mypy per verificare i tipi delle celle Jupyter
#%load_ext nb_mypy
```

In [2…
```
from images import load, visd

# definiamo qualche colore
```

```python
black =   0,   0,   0
white = 255, 255, 255
red   = 255,   0,   0
green =   0, 255,   0
blue  =   0,   0, 255
cyan  =   0, 255, 255
yellow= 255, 255,   0
purple= 255,   0, 255
gray  = 128, 128, 128

# definizione di tipi
Colore   = tuple[int,int,int]
Immagine = list[list[Colore]]

def crea_immagine(larghezza : int, altezza : int, colore : Colore=black) -> Immagine :
    return [  [ colore ]*larghezza
                for i in range(altezza)
            ]

def draw_pixel(img : Immagine, x : int, y : int, colore : Colore) -> None:
    altezza   = len(img)
    larghezza = len(img[0])
    if  0 <= x < larghezza and 0 <= y < altezza:
        img[y][x] = colore

def bound(canale : float|int, m:int=0, M:int=255 ) -> int:
    "trasformo il valore in intero all'interno di [m..M]"
    canale = round(canale)
    return min(max(canale, m), M)

from typing import Callable
Filtro = Callable[[Colore], Colore]   # funzione che accetta un Colore e produce un Colore

from copy import deepcopy
def applica_filtro( img : Immagine, filtro : Filtro ) -> Immagine:
    'creo una nuova immagine in cui ciascun pixel è trasformato con la funzione filtro(Colore)->Colore'
    # copio l'immagine
```

```
    #copia = deepcopy(img)
    copia = [ riga.copy() for riga in img ]
    # tutti i pixel vengono sostituiti con il risultato del filtro
    for y, riga in enumerate(img):
        for x, colore in enumerate(riga):
            copia[y][x] = filtro(colore)   ### QUI eseguo il filtro sul pixel corrente
    return copia

# - devo sapere dove sono nella immagine e avere accesso a tutta l'immagine!
#                     x    y    img       L    A
FiltroXY = Callable[[int, int, Immagine, int, int], Colore]    # funzione filtro che conosce x,y,img,L,A

def applica_filtro_XY( img : Immagine, filtro : FiltroXY ) -> Immagine:
    # ricevo nell'argomento 'filtro' una funzione che calcola
    # per ogni colore e posizione X,Y il nuovo colore
    'applicazione di un filtro che dipende da x,y, dalla immagine e dalle dimensioni W,H'
    W,H = len(img[0]),len(img)
    copia = deepcopy(img)
    for y in range(H):
        for x in range(W):
            copia[y][x] = filtro(x, y, img, W, H)   ### QUI chiamo il filtro
    return copia

img = load('3cime.png')
```

Effetto swirl/vortice

Ruotiamo l'immagine di un angolo che cresce allontanandoci dal centro

Dati:

- centro **x,y** del vortice
- intensità **k** del vortice (quanto si deve ruotare a seconda della distanza dal centro)

Per ogni pixel della immagine

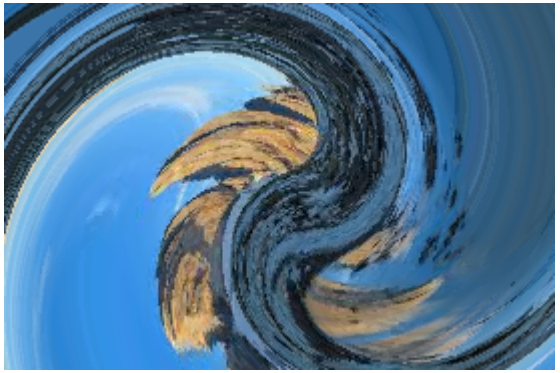
- calcolo la distanza **D** e l'angolo **A** rispetto al centro

- aumento l'angolo di un fattore **$k*D$**
- torno il pixel alla stessa distanza ma angolo aumentato (ma dentro l'immagine)

In [3…
```python
from math import dist, atan2, cos, sin

#         pos  immagine    centro   fattore
def swirl(x,y, img, W, H,  xc, yc,  k):
    """FiltroXY che sposta il pixel ruotandolo attorno a xc,yc con effetto vortice
    - xc,yc: centro del vortice
    - k:     fattore di amplificazione della rotazione
    """
    distanza = dist((x,y),(xc,yc))          # distanza da x,y al centro xc,yc
    dx, dy   = x-xc, y-yc                   # proiezioni del segmento xc,yc -> x,y sugli assi
    angolo   = atan2(dy,dx)                 # arcotangente -> angolo in radianti
    angolo1  = angolo + distanza*k          # aumento l'angolo crescendo con la distanza
    nuovoX   = xc + distanza*cos(angolo1)   # posizione X del nuovo punto
    nuovoY   = yc + distanza*sin(angolo1)   # posizione Y del nuovo punto
    X = bound(nuovoX, 0, W-1)               # resto dentro l'immagine in X
    Y = bound(nuovoY, 0, H-1)               # ed in Y
    return img[Y][X]                        # torno il pixel "ruotato"
```

In [4…
```python
sw1 = applica_filtro_XY(img,
                lambda x, y, img, W, H: swirl(x, y, img, W, H, 150, 100, 0.02) )
visd(sw1)
```



Effetto onda

Voglio distorcere l'immagine **in verticale** per dare un effetto di **onde** più forte in basso e più debole in alto

Dati:

- **N**: quante onde vogliamo nella larghezza della immagine
- **K**: quanto amplifichiamo lo spostamento verticale **dy**

Alziamo la Y di una quantità $k* y * sin(x)$ (minore in alto e maggiore in basso per y più grandi)

Per ogni pixel della immagine

- calcolo l'incremento **dy**
- aumento la y
- torno quel pixel (ma dentro l'immagine)

In [5…
```python
from math import dist, atan2, cos, sin, pi

#        pos  immagine   fattore   num_onde
def wave(x,y, img, W, H, k,        N):
    """FiltroXY che sposta il pixel in verticale con effetto onda
    - k: fattore di amplificazione dello spostamento y
    - N: numero di "onde" nella larghezza della immagine
    """
    x_percentuale = x/W                  # posizione orizzontale in percentuale per ciascun pixel
    x_radianti    = x_percentuale*2*pi   # una onda è larga 2*pi radianti, quindi converto x in radianti
    spostamento   = k*cos(N*x_radianti)  # calcolo lo spostamento come cosinusoide su N onde complete
    dY = y*spostamento                   # moltiplico per y così la parte alta resta più ferma
    Y = bound(y+dY, 0, H-1)              # resto nella immagine
    return img[Y][x]                     # torno il pixel "spostato"
```

In [6…
```python
sw1 = applica_filtro_XY(img,
                        lambda x, y, img, W, H:   # per ottenere il pixel
                            wave(x, y, img, W, H, # applico wave aggiungendo i parametri
                                 0.1,             # k: fattore di amplificazione verticale
                                 3) )             # N: numero di onde nella larghezza della immagine
visd(sw1)
```

PAUSA -> esercizi 13 e 42

Programmazione ad oggetti (OOP)

- Gli oggetti sono la fusione di:
  - **attributi**: i dati che caratterizzano una particolare entità
    (per esempio un cane ha un peso, un nome, un genere, una età, un colore ...)
  - **metodi**: le funzionalità caratteristiche quella particolare entità
    (un cane abbaia, si muove, scodinzola, mangia, morde, si accoppia ...)

La descrizione di una tipologia di oggetto si chiama **classe** (esempio: i Cani)

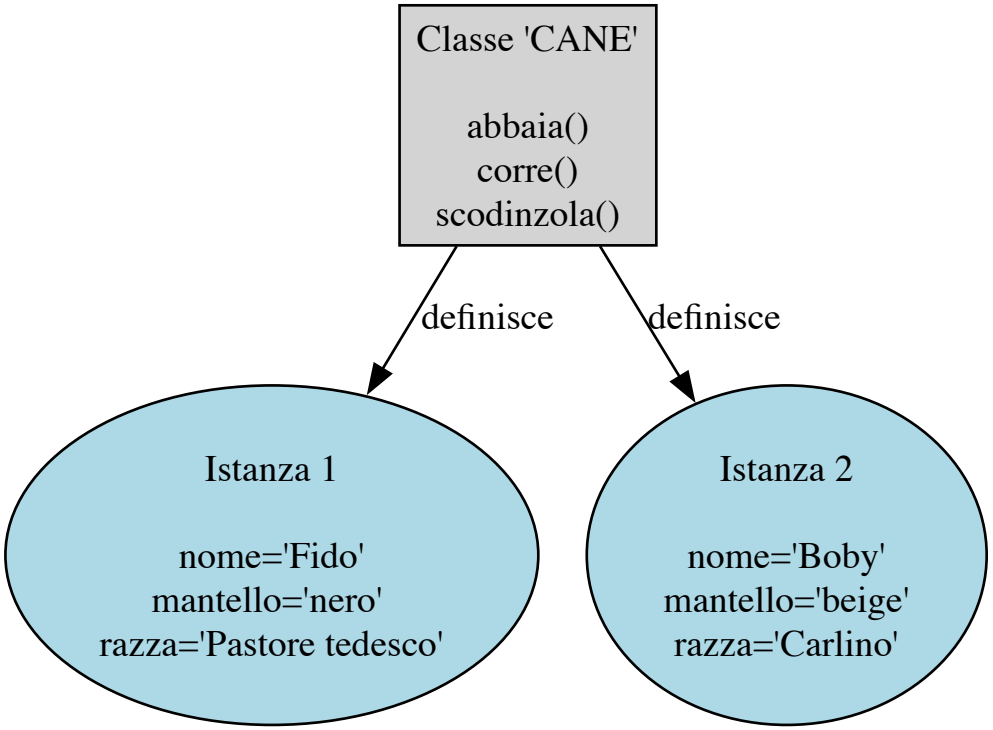
Ciascun individuo di una certa tipologia si chiama **istanza** della **classe** (esempio: Fido)

Abbiamo usato ampiamente gli oggetti (str, int, dict, tuple, float, bool ...) e i loro metodi

In [7…

```
from pygraphviz import AGraph
figura = AGraph(directed=True)
figura.add_node('C',  label="Classe 'CANE'\n\nabbaia()\ncorre()\nscodinzola()",              shape="box
figura.add_node('I1', label="Istanza 1\n\nnome='Fido'\nmantello='nero' \nrazza='Pastore tedesco'", shape
figura.add_node('I2', label="Istanza 2\n\nnome='Boby'\nmantello='beige'\nrazza='Carlino'",        shape
figura.add_edges_from([('C', 'I1'),('C','I2')], label="definisce")
figura.layout(prog='dot')
figura
```

Out [7…



Come definire un nuovo tipo di oggetto (una *classe*)

```
class NomeDellaClasse (EstendendoLaClasse):
    # class variable: valori condivisi tra tutte le istanze/individui
    attributo_di_classe = valore
    ...

    # metodo speciale che inizializza gli attributi dell'istanza
    def __init__(self, <argomenti>):
        # instance variable: definisco un valore personale di un individuo/istanza
        self.attributo_individuale = valore1
        ...

    def metodo1(self, <altri argomenti>):     # comportamenti di tutti gli individui
        corpo del metodo
    ...
```

In [8…

```python
class Cane:
    # class variable: valori condivisi tra tutte le istanze/individui
    colori = ['nero', 'beige', 'bianco', 'a pois']
    razze  = ['pastore tedesco', 'labrador', 'barboncino', 'bulldog', 'carlino']

    # metodo speciale che inizializza gli attributi dell'istanza
    def __init__(self, nome:str, razza:str, colore:str):
        # instance variable: definisco un valore personale di un individuo/istanza
        if not isinstance(nome, str) or len(nome)==0:
            raise ValueError(f'Nome {nome} non valido')
        if razza not in Cane.razze:
            raise ValueError(f'Razza {razza} non valida')
        if colore not in Cane.colori:
            raise ValueError(f'Colore {colore} non valido')
        self.nome = nome
        self.razza = razza
        self.colore = colore

    def abbaia(self):     # comportamenti di tutti gli individui
        pront('Bau')
    def scodinzola(self):
        print('I am happy!')
    def corre(self, velocità):
        print("Zoom!!!")
```

Come creare una "istanza" di una classe (un "individuo")

```python
A = NomeDellaClasse(argomenti_del_metodo_init)

# Esempio: costruisco un insieme con gli elementi di una lista
L = set([1,2,3,4,2,4,1,5,6,7])        # 'set' è il nome della classe degli insiemi
```

In [9…
```
Fido = Cane('Fido', 'pastore tedesco', 'nero')   # creo l'istanza Fido della classe Cane
Boby = Cane('Boby', 'carlino', 'beige')          # creo l'istanza Boby della classe Cane
Diavolo = Cane('Diavolo', 'bulldog', 'a pois')   # creo l'istanza Diavolo della classe Cane

print(Diavolo)
```

```
<__main__.Cane object at 0x105b64d70>
```

Come eseguire il metodo di un oggetto

```
risultato = oggetto.metodo1(argomenti_del_metodo)

# Esempio: cerco la posizione di una stringa in un testo
posizione = testo.find('Paperino')
```

In [1…
```
Diavolo.scodinzola()
```

```
I am happy!
```

Trasformiamo i colori in oggetti

Vogliamo poter (ri)scrivere le trasformazioni che abbiamo fatto sui colori come espressioni semplici che in realtà operano sui tre canali R,G,B

Approfittiamo del fatto che Python converte le espressioni (ad esempio quelle aritmetiche) in chiamate a metodi speciali:

Operatore	Trasformazione			Metodo speciale da definire
`-`	`X - Y`	diventa	`X.__sub__(Y)`	`__sub__(self, other)`
`+`	`X + Y`	diventa	`X.__add__(Y)`	`__add__(self, other)`
`*`	`X * Y`	diventa	`X.__mul__(Y)`	`__mul__(self, k)`
`/`	`X / Y`	diventa	`X.__truediv__(Y)`	`__truediv__(self, k)`
`//`	`X // Y`	diventa	`X.__floordiv__(Y)`	`__floordiv__(self, k)`
`%`	`X % Y`	diventa	`X.__mod__(Y)`	`__mod__(self, k)`

Operatore		Trasformazione		Metodo speciale da definire
`==`	`X == Y`	diventa	`X.__eq__(Y)`	`__eq__(self, other)`
`len(X)`	`len(X)`	diventa	`X.__len__()`	`__len__(self)`

ECCETERA ...

Una matematica dei colori

Vogliamo riscrivere le trasformazioni sui colori come operazioni matematiche

Per esempio:

- **luminosità(colore, k)** vogliamo scriverlo come **colore * k**
- **contrasto(colore,k)** vogliamo scriverlo **(colore-grigio) * k + grigio**

Come?

- basta ridefinire i metodi che realizzano le operazioni matematiche (**`__add__`** , **`__mul__`** , ...)

Per prima cosa definiamo la classe `Colore` ed il suo costruttore

Il metodo **`__init__(self, ...)`** è speciale e serve ad **inizializzare** l'individuo/istanza che stiamo creando

- l'oggetto di base viene creato e fornito nell'argomento **`self`**
- il metodo `__init__` **aggiunge tutti gli attributi** necessari alla istanza

In [1…
```
# libreria che permette di definire una classe spezzata in più celle Jupyter
import jdc
# ATTENZIONE: nella realtà i metodi devono essere INDENTATI dentro la classe
```

In [1…
```
class Colore:   # rappresentazione di un colore RGB
    def __init__(self, R : float, G : float, B : float):
        "un colore contiene i tre canali R,G,B"
        self._R = R     # metto ciascun valore in un attributo della istanza 'self'
```

```
        self._G = G     # per convenzione gli attributi "privati" iniziano con '_'
        self._B = B

    # NOTA: tutti i metodi devono essere INDENTATI
    # DENTRO la classe
```

In [1…
```
A = Colore(123, 234, 12)
A._R, A      # vediamo cosa contiene l'attributo _R e cos'è l'oggetto A
```

Out[1…
```
(123, <__main__.Colore at 0x1046bcbc0>)
```

Definiamo un paio di metodi di comodo

- come convertire un colore in tripla (per poi salvare i file con **images.save**)
- come visualizzare il colore come stringa (`__repr__` oppure `__str__`)

In [1…
```
%%add_to Colore
# trucco del modulo jdc per aggiungere metodi alla classe definendoli in una cella Jupter diversa

def _asTriple(self) -> tuple[int,int,int] :   # C -> (R,G,B)
    "creo la tripla di interi tra 0 e 255 che serve per le immagini PNG"
    # NOTA: SOLO quando devo creare un file PNG mi serve che il pixel sia intero nel range 0..255
    def bound(C):
        return min(255, max(0, int(round(C))))
    return bound(self._R), bound(self._G), bound(self._B)

def __repr__(self) -> str:   # C -> "Colore(R,G,B)"
    "stringa che deve essere visualizzata per stampare il colore"
    # uso una f-stringa che mostra i 3 valori
    return f"Colore({self._R},{self._G},{self._B})"
```

In [1…
```
# Esempio di __repr__
Colore(255,0,255)
```

Out[1…
```
Colore(255,0,255)
```

Poi (ri)definiamo somma e differenza (`__add__` e `__sub__`) tra Colori

In [1…
```
%%add_to Colore
# trucco del modulo jdc per aggiungere metodi alla classe in una cella Jupter diversa

def __add__(self, other : 'Colore') -> 'Colore':  # C1 + C2
    "somma tra due colori"
    assert type(other) == Colore, "Il secondo argomento non è un Colore"
    return Colore(self._R+other._R, self._G+other._G, self._B+other._B)

def __sub__(self, other : 'Colore') -> 'Colore':  # C1 - C2
    "differenza tra due colori"
    assert type(other) == Colore, "Il secondo argomento non è un Colore"
    return Colore(self._R-other._R, self._G-other._G, self._B-other._B)
```

In [1…
```
Colore(255,0,0) + Colore(0,0,255)
```

Out[1…
```
Colore(255,0,255)
```

e prodotto o divisione per una costante K (`__mul__` e `__truediv__`)

In [1…
```
%%add_to Colore
def __mul__(    self, k : float|int) -> 'Colore': # moltiplicazione*k
    "moltiplicazione di un colore per una costante numerica k"
    assert isinstance(k, (int,float)), "Il secondo argomento non è un numero"
    return Colore(self._R*k, self._G*k, self._B*k)

def __truediv__(self, k : float|int) -> 'Colore': # divisione/k
    "divisione di un colore per una costante numerica diversa da 0"
    assert isinstance(k, (int,float)) and k!=0 , "Il secondo argomento non è un numero diverso da 0"
    return Colore(self._R/k, self._G/k, self._B/k)
```

In [1…
```
Colore(2,3,4)*5
```

Out[1…
```
Colore(10,15,20)
```

e un paio di metodi generali

- luminosità media
- grigio

In [2…
```
%%add_to Colore
def luminosità(self) -> float:   # C -> luminosità
    "calcolo la luminosità media di un pixel"
    #L = (self._R + self._G + self._B)/3               # media semplice
    L = 0.299*self._R + 0.587*self._G + 0.114*self._B  # media pesata
    return L

def grigio(self) -> 'Colore':    # C -> grigio
    "creo un colore grigio con la stessa luminosità media"
    L = self.luminosità()
    return Colore(L,L,L)
```

In [2…
```
ROSSO = Colore(255,0,0)
ROSSO.grigio(), ROSSO.luminosità(), ROSSO.grigio()._asTriple()
```

Out[2…
```
(Colore(76.24499999999999,76.24499999999999,76.24499999999999),
 76.24499999999999,
 (76, 76, 76))
```

A questo punto fa comodo avere un po' di colori con nomi "umani"

In [2…
```
# solo dopo che ho completato la definizione della classe Colore
# posso definire dei colori come suoi **attributi di classe*
# che contengono *istanze di Colore* ad esempio alcuni colori standard

# NON VA INDENTATO DENTRO la classe 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.yellow= Colore(255, 255,   0)
Colore.purple= Colore(255,   0, 255)
Colore.cyan  = Colore(  0, 255, 255)
Colore.grey  = Colore.white / 2    # STO USANDO __truediv__ !!!

Colore.grey
```

Out[2… `Colore(127.5,127.5,127.5)`

Introduciamo alcune trasformazioni del pixel

- negativo
- luminosità per **k**
- contrasto cambiato di **k**

In [2…
```
%%add_to Colore
def negativo(self) -> 'Colore':   # C -> inverso
    "ottengo il colore 'inverso'"
    return Colore.white - self

def illumina(self, k : float)  -> 'Colore':  # C-> colore più luminoso/scuro
    'ottengo il colore schiarito/scurito di un fattore k'
    return self * k

def contrasto(self, k : float) -> 'Colore':  # C -> colore più/meno contrastato
    "ottengo il colore allontanato/avvicinato di un fattore k dal grigio"
    return Colore.grey + (self - Colore.grey)*k
```

Esempi

In [2…
```
# Esempi
rosso = Colore(255,  0, 0)
verde = Colore(  0,255, 0)
p3 = rosso + verde # uso l'operatore somma tra due colori che ho definito
```

```
print('somma di', rosso, 'e', verde, 'uguale', p3)
```

```
somma di Colore(255,0,0) e Colore(0,255,0) uguale Colore(255,255,0)
```

In [2…
```
p4 = p3 * 0.5      # uso l'operatore prodotto per una costante che ho definito
print(p3, 'per', 0.5, 'uguale', p4)
```

```
Colore(255,255,0) per 0.5 uguale Colore(127.5,127.5,0.0)
```

In [2…
```
# media di 4 colori
LC = [rosso, verde, p3, p4]
print('media di', LC, 'viene', sum(LC, Colore.black)/len(LC))

# NOTA: se sommiamo oggetti dobbiamo dare il valore iniziale a cui "sommarli", in questo caso il colore
```

```
media di [Colore(255,0,0), Colore(0,255,0), Colore(255,255,0), Colore(127.5,127.5,0.0)] viene Colore(15
9.375,159.375,0.0)
```

In [2…
```
C = Colore(56, 200, 31)
print('luminosità diminuita del 20%', C, 'diventa', C.illumina(0.8))
print('contrasto  aumentato del 50%', C, 'diventa', C.contrasto(1.5))
print('contrasto  diminuito del 20%', C, 'diventa', C.contrasto(0.8))
```

```
luminosità diminuita del 20% Colore(56,200,31) diventa Colore(44.800000000000004,160.0,24.8)
contrasto  aumentato del 50% Colore(56,200,31) diventa Colore(20.25,236.25,-17.25)
contrasto  diminuito del 20% Colore(56,200,31) diventa Colore(70.3,185.5,50.3)
```

In [2…
```
# se trasformo un colore in tripla i tre canali tornano interi tra 0 e 255
Colore(20.25,336.25,-17.25)._asTriple()
```

Out[2… (20, 255, 0)

Definiamo ora una classe Immagine

- che contiene una **lista di liste di Colore** invece che di triple RGB
- e magari conosce anche **le proprie dimensioni**
- che sa applicare **filtri semplici** o **filtri XY**
- che posso **caricare da un file**

- che posso **salvare su un file**
- sulla quale posso **disegnare** (line, pixel, rectangle ...)

Comincio con classe e costruttore `__init__`

Posso creare una immagine in due modi:

- **leggendola da un file PNG** e convertendo le triple in Color
- o fornendo **dimensioni e colore di sfondo**

in entrambi i casi mi segno le dimensioni una volta per tutte

NOTA altri linguaggi permettono di definire più costruttori diversi, Python ne permette uno solo (ma ci sono modi di ovviare)

In [2…

```
import images, os
from math import dist
from typing import Optional, Callable

class Immagine:

    def __init__(self, larghezza : int, altezza : int, sfondo : Optional[Colore]=Colore.black):
        " creo una immagine monocolore e ne ricordo le dimensioni "
        assert (    isinstance(altezza,  int) and altezza > 0
                and isinstance(larghezza,int) and larghezza > 0
                and isinstance(sfondo, Colore)), "parametri sbagliati per creare una immagine vuota"
        self._W   = larghezza
        self._H   = altezza
        self._img = [ [ sfondo for _ in range(self._W) ] for _ in range(self._H) ]

    @classmethod # questo metodo può solo eseguire operazioni sulla classe
    def load(cls, filename : str): # riceve l'oggetto classe invece che l'istanza
        """Costruttore alternativo con un nome diverso che legge l'immagine dal file"""
        assert (os.path.exists(filename)
                and filename.endswith('.png')), f"il file {filename} non esiste oppure non è in formato
        img = images.load(filename)
        W, H = len(img[0]),len(img)
```

```
        myself = cls(W,H)   # creo una immagine di dimensioni WxH di colore nero
        myself._img = [ [ Colore(R,G,B) for R,G,B in riga ] for riga in img ]
        return myself
```

poi definisco un paio di metodi di utilità

- visualizzazione di un oggetto Immagine come stringa (col metodo `__repr__`)
- conversione da Colori a triple
- salvataggio su file
- visualizzazione in Spyder/Jupyter/Ipython

In [3…

```
%%add_to Immagine

def __repr__(self) -> str:  # I -> "Immagine(WxH)"
    "per stampare l'immagine ne mostro solo le dimensioni"
    return f"Immagine({self._W}x{self._H})"

# metodo "privato", inizia per '_'
def _asTriples(self) -> list[list[tuple[int,int,int]]]: # conversione in liste di liste di triple
    "conversione della immagine da matrice di Colore a matrice di triple"
    return [ [c._asTriple() for c in riga] for riga in self._img ]

def save(self, filename : str) -> 'Immagine':   # salvataggio
    "si salva l'immagine dopo averla convertita in matrice di triple"
    images.save(self._asTriples(), filename)
    return self        # torno l'immagine così posso concatenare più operazioni grafiche

def visualizza(self):  # mostra l'immagine in Spyder/Jupyter
    "visualizzo l'immagine in Spyder/Jupyter"
    return images.visd(self._asTriples())
```

In [3…

```
trecime = Immagine.load('3cime.png') # così si esegue un metodo di classe
print(trecime)
trecime.visualizza()
```

```
Immagine(275x183)
```

e un paio di metodi per scrivere o leggere un pixel senza sbordare

In [3…

```
%%add_to Immagine
def set_pixel(self, x: float|int, y: float|int, color : Colore) -> 'Immagine':   # cambio il pixel
    "cambio un pixel solo se è dentro l'immagine"
    x = round(x)      # float -> int
    y = round(y)      # float -> int
    if 0 <= x < self._W and 0 <= y < self._H:
        self._img[y][x] = color
    return self       # torno l'immagine così posso concatenare più operazioni grafiche

def get_pixel(self, x: float|int, y: float|int) -> Colore :  # leggo il pixel più vicino a x,y
    "leggo un pixel se è dentro l'immagine oppure torno il più vicino sul bordo"
    x = round(x)      # float -> int
    y = round(y)      # float -> int
    x = min(self._W-1, max(0, x))
    y = min(self._H-1, max(0, y))
    return self._img[y][x]
```

In [3…

```
print(trecime.get_pixel(100,2000))
for i in range(100):
    trecime.set_pixel(i,i,Colore.green)
trecime.visualizza()
```

```
Colore(52,91,120)
```