Table of Contents

RECAP: Immagini
Programmazione ad oggetti (OOP)
Come definire un nuovo tipo di oggetto
Traformiamo i colori in oggetti
  Per prima cosa definiamo la classe ed il suo costruttore
  Poi (ri)definiamo somma e differenza (`__add__` e `__sub__`)
  e prodotto o divisione per una costante K (`__mul__` e `__truediv__`)
  più un paio di metodi di comodo
e un paio di metodi generali
  A questo punto fa comodo avere un po' di colori con nomi "umani"
  Introduciamo alcune trasformazioni del pixel
Esempi
Definiamo ora una classe Immagine
  Comincio con classe e costruttore `__init__`
  poi definisco un paio di metodi di utilità
  e un paio di metodi per scrivere o leggere un pixel senza sbordare
  dei metodi per disegnare linee
  e dei modi di disegnare figure
  e i meccanismi per applicare un filtro

Fondamenti di Programmazione

Andrea Sterbini

lezione 14 - 21 novembre 2022

RECAP: Immagini

- creazione/load/save
- rotazione
- disegno di linee verticali/orizzontali/diagonali

- disegno di rettangoli ed ellissi
- trasformazione di colori (grigio/negativo/luninosità/contrasto)
- trasformazione tramite filtri con e senza posizione

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** (i Cani)

Ciascun individuo di una certa tipologia si chiama **istanza** della **classe**

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

Come definire un nuovo tipo di oggetto

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

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

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

Traformiamo i colori in oggetti

Per poter (ri)scrivere le trasformazioni che abbiamo fatto sui colori come espressioni semplici

Per esempio:

- **luminosità(colore, k)** diventa **colore*k**
- **contrasto(colore,k)** diventa **(colore-grigio)*k + grigio**

Come?

- basta ridefinire i metodi che realizzano le operazioni matematiche (**`__add__`** , **`__mul__`** , ...)
- così definiamo una **matematica dei colori**

Per prima cosa definiamo la classe ed il suo costruttore

Il metodo **`__init__(self, ...)`** è speciale e serve ad **inizializzare** l'individuo/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 [31]:

```python
class Colore:   # rappresentazione di un colore RGB
    # definisco un elenco di nomi di colori standard, ma li creiamo dopo
    black  : 'Colore'  # i tipi li metto come stringa perchè
    white  : 'Colore'  # la classe Colore non è ancora
    red    : 'Colore'  # completamente definita
    green  : 'Colore'
    blue   : 'Colore'
    cyan   : 'Colore'
    yellow : 'Colore'
    purple : '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

    # NOTA: tutti i metodi devono essere INDENTATI
    # dentro la classe

A = Colore(123, 234, 12)
A._R
```

Out[31]: 123

Poi (ri)definiamo somma e differenza (__add__ e __sub__)

In [33]:

```python
%%add_to Colore
# trucco del modulo jdc per aggiungere metodi alla classe in una cella diversa

def __add__(self, other : 'Colore') -> 'Colore':  # C1 + C2
    "somma tra due colori"
    # FIXME: prima controllo che l'altro sia un colore
    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"
    # FIXME: prima controllo che l'altro sia un colore
    return Colore(self._R-other._R, self._G-other._G, self._B-other._B)


```

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

In [34]:
```
%%add_to Colore
def __mul__(    self, k : float) -> 'Colore': # moltiplicazione*k
    "moltiplicazione di un colore per una costante"
    # FIXME: controllare che k sia float
    return Colore(self._R*k, self._G*k, self._B*k)

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

più un paio di metodi di comodo

- conversione da colore a tripla (per poi salvare i file con **images.save**)
- visualizzazione del colore come stringa (**`__repr__`** oppure **`__str__`**)

In [35]:
```
%%add_to Colore
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 assicuro che il pixel sia intero n
    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})"
```

e un paio di metodi generali

- luminosità media
- grigio

In [36]:
```
%%add_to Colore
def luminosità(self) -> float:   # C -> luminosità
    "calcolo la luminosità media di un pixel (senza badare se è un valore intero)"
    return (self._R + self._G + self._B)/3

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

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

In [37]:
```
# solo dopo che ho definito la classe Color posso definire dei colori
# posso aggiungerle degli *attributi di classe*
# che contengono *istanze di Color*
# ad esempio alcuni colori standard

# NON INDENTATO SOTTO 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[37]:
```
Colore(127.5,127.5,127.5)
```

Introduciamo alcune trasformazioni del pixel

- negativo
- luminosità
- contrasto

In [38]:
```
%%add_to Colore
def negativo(self):    # C -> inverso
    "ottengo il colore 'inverso'"
    return Colore.white-self

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

def contrasto(self, k : float):  # 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 [43]:
```
# 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(p3)

p4 = p3 * 0.5      # uso l'operatore prodotto per una costante che ho definito

#print(p4)

# media di 4 colori
LC = [rosso, verde, p3, p4]
#print(sum(LC, Colore.black)/len(LC))

C = Colore(56, 200, 31)

print(C.illumina(0.8))
print(C.contrasto(1.5))
#print(C.contrasto(0.8))

Colore.contrasto(C, 0.8)
```

```
Colore(44.800000000000004,160.0,24.8)
Colore(20.25,236.25,-17.25)
```

Out[43]:
```
Colore(70.3,185.5,50.3)
```

Definiamo ora una classe Immagine

- che contiene una **lista di liste di Colore** invece che di triple RGB
- e magari 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

In [44]:

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

class Immagine:

    def __init__(self,   larghezza : Optional[int]   =None,
                         altezza   : Optional[int]   =None,
                         sfondo    : Optional[Colore]=None,
                         filename  : Optional[str]   =None):
        # FIXME controllare che i valori siano corretti
        if filename:  # leggo la immagine e la converto in una matrice di Colore e ne
            img = images.load(filename)
            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 monocolore e ne ricordo le dimensioni
            self._W   = larghezza
            self._H   = altezza
            self._img = [ [ sfondo for _ in range(larghezza) ]
                                   for _ in range(altezza)
                        ]
```

poi definisco un paio di metodi di utilità

- visualizzazione della immagine come stringa (`__repr__`)
- salvataggio su file
- conversione da Colori a triple
- visualizzazione in Spyder/Jupyter/Ipython

In [45]:
```
%%add_to Immagine

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

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

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

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

In [46]:
```
trecime = Immagine(filename='3cime.png')
trecime.visualizza()
```

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

In [47]:

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

def get_pixel(self, x: float, y: float) -> Colore :  # leggo il pixel più vicino
    "leggo un pixel se è dentro l'immagine oppure torno il più vicino sul bordo"
    x = int(round(x))       # float -> int
    y = int(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]
```

dei metodi per disegnare linee

- orizzontale
- verticale
- inclinata (qualsiasi)

In [48]:

```
%%add_to Immagine
def draw_line_H(self, x: int, y: int, x1: int, color: Colore) -> 'Immagine': # linea 

def draw_line_V(self, x: int, y: int, y1: int, color: Colore) -> 'Immagine':  # linea

def draw_line(self, x1: int,y1: int, x2:int,y2:int, color: Colore) -> 'Immagine':   #
    "disegno una linea qualsiasi"
    dx = x1-x2
    dy = y1-y2
    if dx == 0:                    # se dx=0  mi muovo in verticale
        self.draw_line_V(x1,y1,y2)
    elif dy == 0:                  # se dy=0  mi muovo in orizzontale
        self.draw_line_H(x1,y1,x2)
    elif abs(dx) > abs(dy):        # se dx più grande itero sulle X
        m = dy/dx
        # FIXME: direzione da x1 a x2
        for X in range(x1,x2+1):
            Y = m*(X-x1) + y1
            self.set_pixel(X,Y,color)
    else:                          # altrimenti sulle Y
        m = dx/dy
        # FIXME: direzione da y1 a y2
        for Y in range(y1,y2+1):
            X = m*(Y-y1) + x1
            self.set_pixel(X,Y,color)
    return self
```

e dei modi di disegnare figure

- rettangoli vuoti o pieni
- triangoli vuoti?
- poligoni regolari?
- ellissi e cerchi

In [49]:

```
%%add_to Immagine
def draw_rectangle_full(self, x: int, y: int, x1: int, y1: int, color: Colore) -> 'Im

def draw_rectangle(self, x: int, y: int, x1: int, y1: int, color: Colore) -> 'Immagin

def draw_ellipse_full(self, x1: int,y1: int, x2: int,y2: int, D: int, color: Colore )

def draw_ellipse(self, x1: int,y1: int, x2: int,y2: int, D: int, color: Colore ) -> '

def draw_circle_full(self, xc: int, yc: int, r: int, color: Colore) -> 'Immagine':
    "un cerchio è una ellisse con i due fuochi coincidenti e D=2*r"
    return self.draw_ellipse_full(xc, yc, xc, yc, 2*r, color)

def draw_circle(self, xc: int, yc: int, r: int, color: Colore) -> None:
    "un cerchio è una ellisse con i due fuochi coincidenti e D=2*r"
    return self.draw_ellipse(xc, yc, xc, yc, 2*r, color)


```

e i meccanismi per applicare un filtro

- **applica_filtro** (solo al singolo pixel)
- **applica_filtro_XY** (per filtri che devono conoscere la posizione)

In [50]:

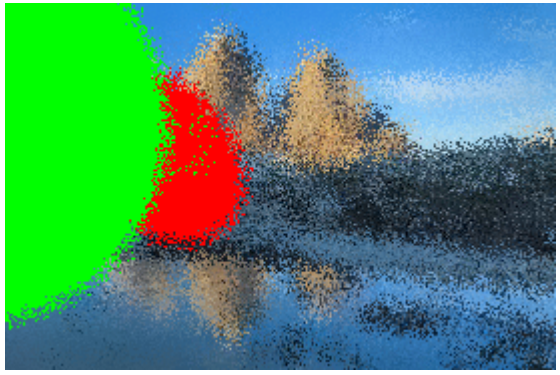
```
%%add_to Immagine
def applica_filtro(self, filtro : Callable[[Colore], Colore]) -> 'Immagine':↔

def applica_filtro_XY(self,
                      filtro : Callable[[int, int, 'Immagine'], Colore]) -> 'Immagine
    "creo una nuova immagine applicando a tutti i pixel il filtro XY"
    # non c'è bisogno di passare W,H perchè sono già nella immagine
    nuova = Immagine(self._W, self._H, Colore.black)
    for y in range(self._H):
        for x in range(self._W):
            nuova._img[y][x] = filtro(x,y,self)
    return nuova


```

In [60]:

```
# ESEMPI

img  = Immagine(larghezza=50, altezza=100, sfondo=Colore.red)
A = Immagine(filename='3cime.png')

#img.visualizza()
A.visualizza()

A.draw_ellipse(50,50, 100, 100, 100, Colore.red)
A.visualizza()
A.draw_rectangle(20,60, 100, 80, Colore.green)
A.visualizza()
A.draw_circle(-20,60, 100, Colore.green)
A.visualizza()
#img.draw_circle(10,60, 10, Colore.green).visualizza()
```

In [57]:
```
# per pixellare una immagine su una dimensione S↔
```

In [58]:
```
from random import randint

def rumore(x, y, I, k):          # leggo un pixel a caso nell'intorno [-k, k]
    "sostituisco il pixel con un vicino"
    X = x + randint(-k,k)
    Y = y + randint(-k,k)
    return I.get_pixel(X,Y)

img2.applica_filtro_XY(lambda x,y,I: rumore(x,y,I,5)).visualizza()
```



In [ ]:
```
# per aggiungere rumore casuale ad una immagine
    # possiamo aggiungere a ciascun pixel un piccolo valore random (filtro locale)
```

In [ ]:
```
# per dare l'effetto lente
    # nella zona della lente
    # mettiamo dei pixel che stanno a distanza K volte
    # quella che si ha dal centro della lente
```

In [ ]:

```
# TODO: generalizzare le operazioni di disegno

# Definiamo una classe FiguraGeometrica con i metodi (da specializzare)
    # draw(x, y, Immagine) che usa SOLO Immagine.set_pixel
    # area()               che ne calcola l'area
    # ...

# Definiamo le figure
    # Punto
    # Linea
    # Rettangolo
    # Triangolo
    # PoligonoRegolare
        # Quadrato
        # TriangoloEquilatero
        # Pentagono
    # Ellisse
        # Cerchio
```