lezione14

November 9, 2023

1 Fondamenti di Programmazione

Andrea Sterbini

lezione 14 - 9 novembre 2023

2 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

3 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

4 Come definire un nuovo tipo di oggetto

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

    def __init__(self, <argomenti>):          # metodo speciale che inizializza l'istanza
        self.attributo_individuale = valore1  # instance variable: definisco un valore persona
        ...
```

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

5 Trasformiamo i colori in oggetti

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

Approfittiamo del fatto che Python converte le espressioni aritmetiche in chiamate a metodi speciali: - **`__add__`, `__sub__`, `__mul__`, `__truediv__`, `__eq__`, `__len__`, ...** - che corrispondono agli operatori **`+`, `-`, `*`, `/`, `==`, `len`, ...**

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__`**, ...) - così definiamo una **matematica dei colori**

5.1 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 ricevuto nell'argomento **`self`** - il metodo **`__init__` aggiunge tutti gli attributi** necessari alla istanza

[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
```

[2]:
```
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
        self._B = B

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

[3]:
```
A = Colore(123, 234, 12)
A._R, A      # vediamo cosa contiene l'attributo _R e cos'è l'oggetto A
```

[3]:
```
(123, <__main__.Colore at 0x104216c50>)
```

5.2 Definiamo 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__`**)

[4]:
```
%%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})"
```

[5]:
```
# Esempio di __repr__
Colore(255,0,255)
```

[5]:
```
Colore(255,0,255)
```

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

[6]:
```
%%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)
```

[7]:
```
Colore(255,0,0) + Colore(0,0,255)
```

[7]:
```
Colore(255,0,255)
```

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

[8]:
```
%%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)
```

[9]:
```
Colore(2,3,4)*5
```

[9]:
```
Colore(10,15,20)
```

6 e un paio di metodi generali

- luminosità media
- grigio

[10]:
```
%%add_to Colore
def luminosità(self) -> float:   # C -> luminosità
    "calcolo la luminosità media di un pixel (senza badare se viene 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)
```

[11]:
```
ROSSO = Colore(255,0,0)
ROSSO.grigio(), ROSSO.luminosità()
```

[11]:
```
(Colore(85.0,85.0,85.0), 85.0)
```

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

[12]:
```
# solo dopo che ho completato la definizione della classe Colore
# posso definire dei colori come suoi attributi
# aggiungendo degli *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
```

[12]: `Colore(127.5,127.5,127.5)`

6.2 Introduciamo alcune trasformazioni del pixel

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

[13]:
```
%%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
```

7 Esempi

[14]:
```
# 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)
```

[15]:
```
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)
```

[16]:
```
# media di 4 colori
LC = [rosso, verde, p3, p4]
print('media di', LC, 'viene', sum(LC, Colore.black)/len(LC))
```

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

[17]:
```
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)
```

[18]:
```
# se trasformo un colore in tripla i tre canali tornano interi tra 0 e 255
Colore(20.25,236.25,-17.25)._asTriple()
```

[18]:
```
(20, 236, 0)
```

8 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 ...)

8.1 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

[19]:
```
import images, os
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]=Colore.black, # se non␣
 ↪indicato, lo sfondo è nero
```

```
                        filename  : Optional[str]   =None):
        if filename:  # leggo la immagine e la converto in una matrice di␣
↪Colore e ne ricordo le dimensioni
            assert (os.path.exists(filename)
                    and filename.endswith('.png')), f"il file {filename} non␣
↪esiste oppure non è in formato PNG"
            img = images.load(filename)
            self._W   = len(img[0])
            self._H   = len(img)
            self._img = [ [ Colore(R,G,B) for R,G,B in riga ] for riga in img ]
        else:   # altrimenti 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) ]
```

8.2 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

[20]:
```
%%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())
```

[21]:
```
trecime = Immagine(filename='3cime.png')
print(trecime)
trecime.visualizza()
```

```
Immagine(275x183)
```

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

[22]:
```
%%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]
```

[23]:
```
print(trecime.get_pixel(100,2000))
for i in range(100):
    trecime.set_pixel(i,i,Colore.green)
trecime.visualizza()
```

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

8.4 un metodo per disegnare linee in qualsiasi direzione

- inclinata (qualsiasi)

[24]:
```
%%add_to Immagine
def draw_line(self, x1: int,y1: int, x2:int,y2:int, color: Colore) ->␣
 ↪'Immagine':   # linea qualsiasi
    "disegno una linea qualsiasi"
    if x1>x2: x1,x2 = x2, x1    # riordino le coordinate x
    if y1>y2: y1,y2 = y2, y1    # riordino le coordinate y
    dx = x2-x1                  # proiezione su asse x
    dy = y2-y1                  # proiezione su asse y
    if dx > dy:                 # se dx è più grande itero sulle X
        m = dy/dx
        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
        for Y in range(y1,y2+1):
            X = m*(Y-y1) + x1
            self.set_pixel(X,Y,color)
    return self                # torno l'immagine così posso concatenare più␣
 ↪operazioni grafiche
```

[25]:
```
trecime.draw_line(100,100,50,200,Colore.red).visualizza()
```

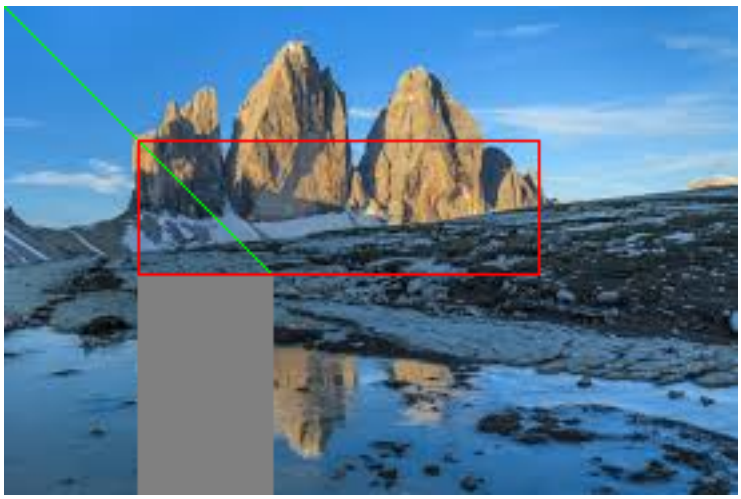
8.5 e dei modi di disegnare figure

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

[26]:
```
%%add_to Immagine
def draw_rectangle_full(self, x: int, y: int, x1: int, y1: int, color: Colore)␣
 ↪-> 'Immagine':   # rettangolo pieno
    "disegno un rettangolo pieno disegnando tante linee orizzontali"
    if y>y1: y,y1 = y1, y    # riordino le coordinate y
    for Y in range(y, y1+1):
        self.draw_line(x,Y,x1,Y, color)
    return self              # torno l'immagine così posso concatenare più␣
 ↪operazioni grafiche

def draw_rectangle(self, x: int, y: int, x1: int, y1: int, color: Colore) ->␣
 ↪'Immagine':   # rettangolo
    "disegno un rettangolo vuoto (4 linee)"
    self.draw_line(x ,y ,x1,y ,color) # sopra
    self.draw_line(x ,y1,x1,y1,color) # sotto
    self.draw_line(x ,y ,x ,y1,color) # a sinistra
    self.draw_line(x1,y ,x1,y1,color) # a destra
    return self              # torno l'immagine così posso concatenare più␣
 ↪operazioni grafiche
```

[27]:
```
trecime.draw_rectangle_full(100,100,50,200,Colore.grey).
 ↪draw_rectangle(50,50,200,100,Colore.red).visualizza()
```



[28]:
```
%%add_to Immagine
def draw_ellipse_full(self, x1: int,y1: int, x2: int,y2: int, D: int, color:␣
 ↪Colore ) -> 'Immagine':
    "una ellisse piena"
    for x in range(self._W):
        for y in range(self._H):
            D1 = dist((x,y),(x1,y1))
            D2 = dist((x,y),(x2,y2))
            if D1+D2 < D:
                self.set_pixel(x,y,color)
    return self                 # torno l'immagine così posso concatenare più␣
 ↪operazioni grafiche

def draw_ellipse(self, x1: int,y1: int, x2: int,y2: int, D: int, color: Colore␣
 ↪) -> 'Immagine':
    "una ellisse vuota"
    for x in range(self._W):
        for y in range(self._H):
            D1 = dist((x,y),(x1,y1))
            D2 = dist((x,y),(x2,y2))
            if abs(D1+D2 - D) < 0.5:
                self.set_pixel(x,y,color)
    return self                 # torno l'immagine così posso concatenare più␣
 ↪operazioni grafiche
```

[29]:
```
Immagine(filename='3cime.png').draw_ellipse_full(40,40,90,90, 80, Colore.red).
 ↪draw_ellipse(50,50,150,30, 180, Colore.green).visualizza()
```

[30]:
```
%%add_to Immagine
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)
```

[31]:
```
Immagine(filename='3cime.png').draw_circle_full(40,40,50, Colore.red).
 ↪draw_circle(150,50,70, Colore.green).visualizza()
```

8.6 e i meccanismi per applicare un filtro

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

[32]:
```
%%add_to Immagine
def applica_filtro(self, filtro : Callable[[Colore], Colore]) -> 'Immagine':
    "creo una nuova immagine applicando a tutti i pixel il filtro"
    nuova = Immagine(self._W, self._H)
    for y,riga in enumerate(self._img):
        for x,pixel in enumerate(riga):
            nuova._img[y][x] = filtro(pixel)
    return nuova

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)
    for y in range(self._H):
        for x in range(self._W):
            nuova._img[y][x] = filtro(x,y,self)
    return nuova
```

[33]:
```
# per pixellare una immagine su una dimensione S
    # costruisco una scacchiera di passo S
    # ogni quadrato ha il colore di:
        # il pixel centrale
        # oppure la media dei suoi pixel
def pixella(x : int,y : int, I : Immagine, S : int) -> Colore :
    X = x - x%S + S//2        # centro del quadratino
    Y = y - y%S + S//2        # che contiene il punto x,y
    return I.get_pixel(X,Y)   # se sbordo ci pensa da solo

def pixella10(x,y,I):    # potremmo definire una funziona ad hoc
    return pixella(x,y,I,10)

# oppure usare una lambda
img2 = Immagine(filename='3cime.png')
img2.applica_filtro_XY(lambda x,y,I: pixella(x,y,I,10)).visualizza()
```

[34]:
```
from random import randint

def rumore(x:int, y:int, I:Immagine, k:int):        # 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()
```

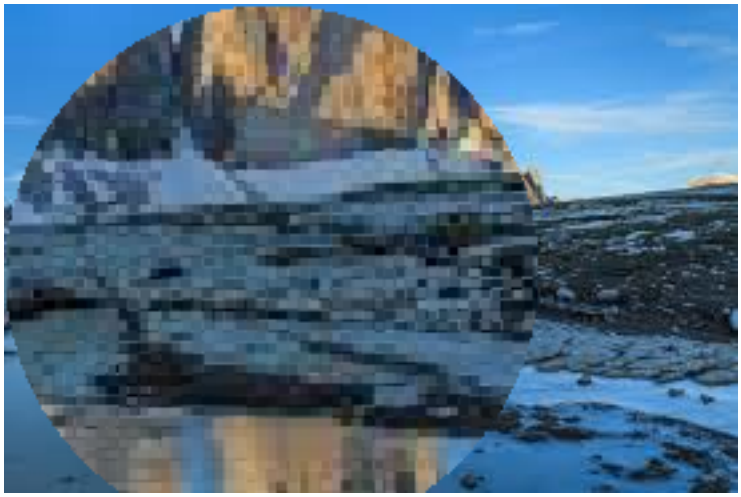
[35]:
```
# per aggiungere rumore casuale ad una immagine
```

```
    # possiamo aggiungere a ciascun pixel un piccolo valore random (filtro␣
 ↪locale)
def color_noise(C:Colore, k:int):
    return C + Colore(randint(-k,k),randint(-k,k),randint(-k,k)) # aggiungo un␣
 ↪colore casuale

img2.applica_filtro(lambda C: color_noise(C,20)).visualizza()
```

[36]:
```
# 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
def lente(x:int, y:int, I:Immagine, xr:int, yr:int, R:int, k:float) -> Colore:
    D = dist((x,y), (xr,yr))
    if D < R:                # se siamo nella lente
        x = xr+(x-xr)*k      # ci spostiamo di k dal centro sulle x
        y = yr+(y-yr)*k      # ci spostiamo di k dal centro sulle y
    return I.get_pixel(x,y) # e torniamo il pixel

Immagine(filename='3cime.png').applica_filtro_XY(lambda x,y,I:␣
 ↪lente(x,y,I,100,100,100,2)).visualizza()
Immagine(filename='3cime.png').applica_filtro_XY(lambda x,y,I:␣
 ↪lente(x,y,I,100,100,100,0.5)).visualizza()
```



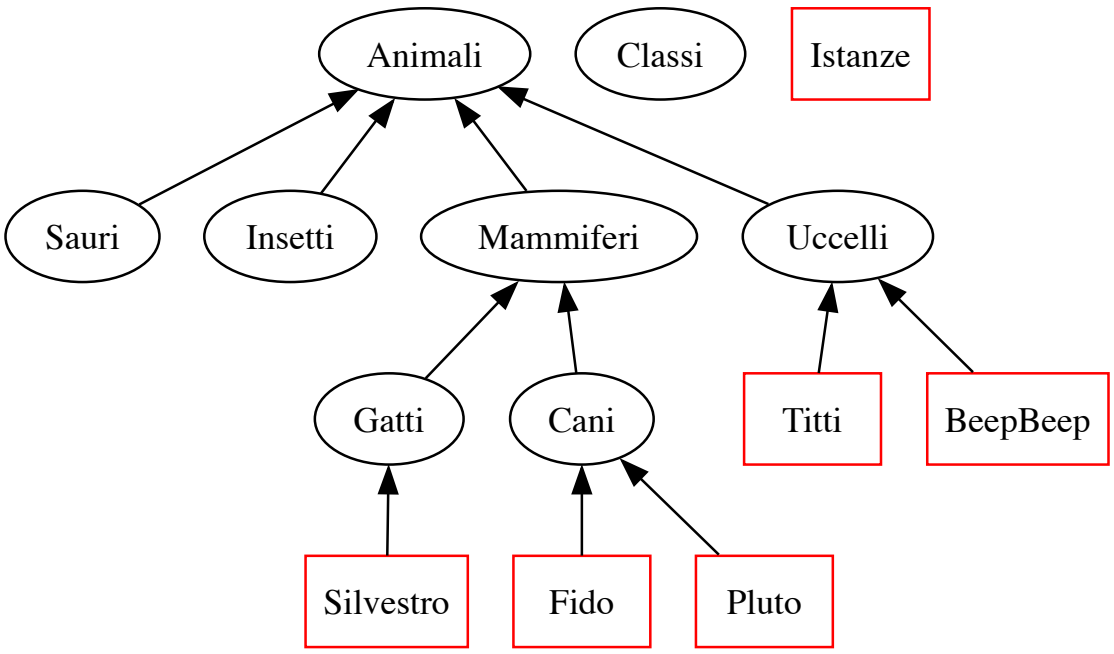
8.6.1 Ereditarietà

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

[37]:
```
from pygraphviz import AGraph
G = AGraph(directed=True, rankdir='TD')
G.edge_attr['dir'] = 'back'
G.
 ↪add_nodes_from(['Titti','Silvestro','Fido','Pluto','BeepBeep'],shape='rect',color='red')
G.add_edge('Animali','Sauri',)
G.add_edge('Animali','Insetti',)
```

```
G.add_path(['Animali','Mammiferi','Gatti','Silvestro',])
G.add_path(['Animali','Mammiferi','Cani','Fido',])
G.add_path(['Animali','Uccelli','Titti',])
G.add_edge('Cani','Pluto',)
G.add_edge('Uccelli','BeepBeep',)
G.add_node('Classi')
G.add_node('Istanze',shape='rect',color='red')
G.subgraph(['Titti','Silvestro','Fido','Pluto','BeepBeep','Animali',
            'Sauri','Insetti','Mammiferi','Uccelli','Gatti','Cani',])
G.subgraph(['Classi','Istanze'])
G.layout('dot')
G
```

[37]:



[38]:

```
# TODO: generalizzare le operazioni di disegno ed i filtri (prox lezione)

# Definiamo i filtri come classi che conoscono come devono comportarsi sulla␣
 ↪Immagine
# FiltroAstratto
    # Filtro
        # Grey
        # Negativo
        # ...
    # FiltroXY
        # Blur
        # Lente
```

```
# 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 la gerarchia di figure
    # Punto
    # Linea
    # Rettangolo
        # Quadrato
    # Triangolo
    # PoligonoRegolare
        # Quadrato
        # TriangoloEquilatero
        # Pentagono
    # Ellisse
        # Cerchio
```