

Visione artificiale

Alessandro Pioggia, Luca Rengo, Federico Brunelli, Leon Baiocchi

3 marzo 2022

Indice

1	Intro	4
1.1	Che cosa vede un computer?	4
1.2	Perchè è difficile?	4
1.3	Perchè è importante?	4
1.4	Possibili applicazioni - Panoramica generale	5
1.5	Strumenti di lavoro	5
1.5.1	Python	5
1.5.2	OpenCV	5
1.5.3	NumPy	5
1.5.4	OpenCV-Python	5
1.5.5	Jupyter Notebook	6
2	Python	7
2.1	Descrizione	7
2.2	Fun fact	7
2.3	Zucchero sintattico	7
2.3.1	Indentazione	7
2.3.2	Variabili	8
2.3.3	Tipi di dati	8
2.3.4	Oggetti, valori, tipi	9
2.3.5	Alcuni oggetti predefiniti	10
2.4	Numeri	10
2.5	Stringhe	10
2.6	Booleans	10
2.7	Operatori aritmetici, di assegnamento, di identità, di appartenenza e bit a bit	11
2.8	Condizioni if..elif..else	11
2.9	Cicli	11
2.10	Liste	12
2.11	Tuple	13
2.12	Slicing	14
2.13	Insiemi	15
2.14	Dizionari	15
2.15	Funzioni	16
2.16	Parametri delle funzioni	16
2.17	Unpacking	17
2.18	Funzioni come oggetti e lambda	18
2.19	Funzioni predefinite	18
2.20	Moduli python	19

3 NumPy	20
3.1 Introduzione	20
3.2 Vantaggi	20
3.3 OpenCV e numPy	20
3.4 La classe ndarray	20
3.5 Creare un array numpy	21
3.5.1 la funzione array	21
3.5.2 Altri modi per costruire un array	22
3.6 Operazioni di base	23
3.7 Operazione prodotto	24
3.8 Operatori di assegnamento	25
3.9 Type cast	26
3.10 Alcune operazione su un array	26
3.11 Funzioni universali (ufunc)	27
3.12 Indicizzazione e slicing su array monodimensionali	27
3.13 Indicizzazione e slicing su array multidimensionali	28
3.14 Iterare un array	29
3.15 Slicing con ellissi	30
3.16 Modifica della forma	30
3.16.1 Altri modi per modificare la forma	31
3.17 Concatenare array	32
3.18 Copie e viste di array	33
3.19 Broadcasting	34
3.20 Indicizzare con un array di indici	34
3.21 Indicizzare un array multidimensionale con un array di indici	35
3.22 Indicizzare con array di booleani	36
4 Immagini	37
4.1 Le immagini digitali	37
4.1.1 Caratteristiche di una immagine	37
4.1.2 Immagine greyscale con punti di luce	37
4.1.3 Coordinate dei pixel	37
4.1.4 Organizzazione dei pixel in memoria	38
4.2 OpenCV	38
4.3 Immagini a colori: tensori 3D	40
4.3.1 Modello RGB	40
4.3.2 Coordinate dei pixel	40
4.3.3 Ordine dei canali	40
4.3.4 Quale è l'ordine dei pixel in memoria?	40
4.4 Caricamento di immagini a colori	41
4.5 Array numpy tridimensionali	42
4.6 Istruzione shape	43
4.7 Rappresentazioni HSV HSL	43
4.7.1 Differenze HSV e HSL	43
4.7.2 Modifica ai canali HSL	44
4.7.3 HSV e HSL in python/openCV	44
4.8 Istogramma di una immagine grayscale	46
4.9 Calcolo istogramma	46
4.10 Analisi dell'istogramma	48
4.11 Operazioni sui pixel	48

4.12	Variazione luminosità e contrasto	48
4.13	Gamma correction - funzione non lineare	49
4.13.1	Formula	49
4.14	Lookup table (LUT)	49
4.15	LUT in python/numpy/OpenCV	50
4.16	LUT da grayscale ad RGB	51
4.17	Operazioni aritmetiche fra immagini	51
4.17.1	Differenza	51
4.17.2	Operazioni bitwise	52
4.17.3	Alpha blending o aliasing	52
4.18	Binarizzare un'immagine grayscale in opencv	53
4.18.1	Soglia globale	53
4.18.2	Soglia locale	53
4.18.3	Binarizzazione su OpenCV	54
4.19	Contrast stretching	55
4.20	Contrast stretching in Python/OpenCV	56
4.21	Equalizzazione in OpenCV	56
4.21.1	Funzionamento	56
4.21.2	Formula	57
5	Calibrazione	58
6	Filtri	59
7	Analisi	60
8	Movimento	61
9	Riconoscimento	62

Capitolo 1

Intro

Idea = cercare di dotare le macchine della vista, si cerca di insegnare ai computer come interpretare le informazioni presenti in immagini e video.

1.1 Che cosa vede un computer?

Un pc vede una matrice di pixel ed ogni pixel è rappresentato da un numero. Questo è il punto di partenza che ci permetterà di capire che cosa c'è nell'immagine.

1.2 Perchè è difficile?

Punti di vista diversi, occlusione (un oggetto ne copre un altro), distorsione, movimento (a volte può tornare utile), variazioni-intra-classe, sfondo complesso (sfondo non omogeneo, pieno di informazioni che non ci interessano).

1.3 Perchè è importante?

Ha molteplici applicazioni, quali

- Sicurezza stradale;
- Salute;
- Prevenzione del crimine;
- Protezione civile;
- Divertimento;
- Domotica.

1.4 Possibili applicazioni - Panoramica generale

- Misurazione, conteggio, qualità : Hanno in comune il fatto di volere misurare la qualità di un qualche oggetto;
 - Misurazione precisa non a contatto : In campo industriale pensiamo ad una catena di montaggio in cui passano prodotti che hanno bisogno di un controllo qualità, che viene svolto da una macchina che sfrutta tecniche di visione artificiale (Controllo delle quantità di liquido in una bottiglietta).
 - Conteggio di veicoli e persone in una piazza, in generale stime dimensionali;
- Elaborazione immagini avanzata
 - Miglioramento immagini;
 - Immagini mediche;
 - Segmentazione di aree agricoli, utili per individuare le immagini aeree e satellitari.
- Riconoscimento
 - Classificazione/individuazione di oggetti : Tag inseriti all'interno dell'immagine, Ricerca per somiglianza di immagini, riconoscere segnali stradali;
 - Riconoscimento persone : punta non solo a classificare, bensì riconoscere univocamente una persona.
- Movimento
 - Video sorveglianza : individuare un ladro, ritrovare uno zaino perso;
 - Navigazione e guida autonoma.

1.5 Strumenti di lavoro

1.5.1 Python

Standard di fatto per lo sviluppo di software scientifico, tra le altre cose anche di visione artificiale. Linguaggio di alto livello, che ha un certo livello di astrazione, in generale con poche linee, se utilizzate in modo corretto possiamo esprimere concetti piuttosto complessi.

1.5.2 OpenCV

OpenCV è la libreria standard per quanto riguarda la visione artificiale, è sviluppata in C++, però è utilizzata anche da altri linguaggi (quali python).

1.5.3 NumPy

La Lazzaro ce lo ha già spiegato.

1.5.4 OpenCV-Python

Una volta richiamato OpenCV da python, tutte le strutture dati (vettori, punti nelle immagini, ecc.) sono array numPy.

1.5.5 Jupyter Notebook

Ambiente web, in cui vengono scritti piccoli pezzi di codice, immagini e altro al fine di creare dei documenti interattivi.

Capitolo 2

Python

Python è il Leonardo Di Caprio dei linguaggi di programmazione. Ma perchè?

2.1 Descrizione

- Python è un linguaggio ad alto livello, general purpose, può essere utilizzato per qualunque tipo di applicazione;
- il codice deve essere facile da leggere, è una prerogativa;
- occorrono poche linee di codice per sviluppare anche concetti complessi;
- supporta programmazione : oop, imperativa e funzionale;

2.2 Fun fact

L'autore, Guido Van Rossum, sceglie il nome python perchè amante del gruppo comico Monty Python, attivo negli anni 70-80.

2.3 Zucchero sintattico

2.3.1 Indentazione

```
#Non si puo' cambiare indentazione in un blocco
print("Salve Cesena")
    print("Salve di nuovo!")

#L'indentazione definisce i blocchi di codice
if 42 < 0:
    print("ciao")
print("ciao ma in corsivo")

#andata a capo
istruzione_molto_lunga = 2 + 5 \
    + 6 + 7

#Piu' istruzioni in una singola riga
print("a"); print("b")
```

2.3.2 Variabili

```
#Le variabili in python non si dichiarano, vengono create automaticamente al  
momento dell'inizializzazione sono case sensitive, iniziano con una lettera  
o underscore e i caratteri ammessi sono le Lettere, i numeri e l'underscore  
(codifica UNICODE)  
  
x = 4  
y = "Visone artificiale"  
print(x, y)  
  
#Assegnamento dello stesso valore a piu' variabili  
  
x = y = z = 90  
  
#Variabili globali e locali  
  
a = "cool" #Variabile globale  
  
def function():  
    a = "bad" #Variabile locale alla funzione  
  
#Se aggiungo global la variabile globale a, definita inizialmente verrà  
#modificata  
def function():  
    global a = "bad"
```

2.3.3 Tipi di dati

Python è tipizzato, solo che, una volta inizializzata una variabile non abbiamo un modo per specializzare un tipo, una volta assegnato un valore viene definito automaticamente (se scrivo tra "", l'oggetto diventerà una stringa). La variabile non ha tipo, è l'oggetto creato che ha un tipo. Il tipo non è legato alla variabile ma all'oggetto!!! I tipi predefiniti di python sono:

- Testo : str
- Numeri : int, float, complex
- Sequenze : list, tuple, range
- Dizionari : dict
- Insiemi : set, frozenset
- booleani : bool
- binari : bytes, bytearray, memoryview

```

x = "ciao" #stringa
x = 20 #int
x = 20.5 #float
x = 1j #numero complesso
x = ["ciao", "come", "stai"] #list
x = ("ciao", "come", "stai") #tuple
x = range(5) #range
x = {"nome" : "Va", "codice" : 17633} #dict
x = {"ciao", "amico"} #set
x = frozenset(x) #frozenset, ovvero un set non modificabile
x = True #bool
x = b"ABCD" #bytes
x = bytearray(5) #bytearray
x = memoryview(bytes(5)) #memoryview
#con type viene stampato il tipo della variabile
type(x)

```

2.3.4 Oggetti, valori, tipi

Qualunque dato in python è un oggetto, ogni oggetto è caratterizzato da : un tipo, un'identità ed un valore. Inoltre:

- il tipo determina le operazioni che l'oggetto supporta;
- l'identità non cambia mai;
- Il tipo determina se un oggetto è mutabile o meno;
- le variabili sono riferimenti ad oggetti.

Python in sè per sè è lento, perchè però ha questo successo? Perchè in un programma sono poche le componenti che devono essere efficienti e sono spesso algoritmi. Gli algoritmi verranno presi da NumPy o altre librerie, che hanno la loro implementazione in C o C++, garantendo massima efficienza.

```

#L'assegnamento crea un oggetto in memoria di tipo Float, a cui la variabile fa
    riferimento
answer = 3.4
print('Type:', type(answer))
print('Identity', id(answer))
print('Value', answer)

#L'assegnamento copia il riferimento dell'oggetto nella nuova variabile
spam = answer
print(spam.equals(answer)) #ritornerà True

# L'istruzione seguente NON modifica l'oggetto ma crea un nuovo oggetto
    contenente il risultato e ne assegna il riferimento alla variabile: si può
    osservare infatti che l'identità dell'oggetto associato a 'answer' è
    cambiata, mentre l'identità di spam è la stessa
answer *= 2
print('Identità nuovo oggetto:', id(answer))
print('Identità vecchio oggetto :', id(spam))

```

2.3.5 Alcuni oggetti predefiniti

Il loro tipo non fa parte dei tipi di base visti prima, hanno un valore particolare.

- `None` : `None` è l'unico oggetto della classe `NoneType`, è simile a `null` ma c'è una differenza sostanziale: quando una variabile è a `null` significa che non ha un riferimento ad un oggetto mentre il `None` è un oggetto che esiste in memoria (ne esiste solo uno). Ponendo una variabile = `None`, la variabile punta a quell'oggetto.
- `Ellipsis` : Sono i puntini di sospensione, è utile in NumPy
- `NotImplemented` : Metodi numerici e di confronto possono restituire questo valore se non implementano l'operazione per determinati operandi.

```
#Sintassi per oggetto Ellipsis
x = ...
y = Ellipsis
print(x, y)
#Confronto fra stringa e numero, restituisce NotImplemented
ret = 'testo'._eq_(42)
```

2.4 Numeri

I numeri possono essere rappresentati attraverso `int`, `float` e `Complex`, possono essere convertiti utilizzando i relativi costruttori (es: `float()`).

2.5 Stringhe

Tutti i caratteri messi dentro ad una stringa con triplici apici vengono considerati, non succede la stessa cosa con apici doppi o singoli. Non esiste il tipo carattere! E' possibile accedere alle stringhe come se fossero array(liste python).

2.6 Booleani

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
a = 200
b = 33
if b > a:
    print("b > a")
else:
    print("b <= a")
#Valori convertiti a True
x = "VA"
y = 15
print(bool(x), bool(y), bool("abc"), bool(123))print(bool(["rosse", "verde",
    "blu"])) #Valori convertiti a False
print(bool(False), bool(None), bool(0), bool(""))print(bool(()), bool([]),
    bool({}))
```

2.7 Operatori aritmetici, di assegnamento, di identità, di appartenenza e bit a bit

Osservazione sulla divisione : la divisione fra numeri interi avviene attraverso il // (risultato arrotondato all'intero inferiore). Invece usando / ritornerà un float.

Per il resto si trova tutto sulle slide del prof oppure sulla documentazione ufficiale di python.

2.8 Condizioni if..elif..else

```
a = 200
b = 1
if b > a:
    print("b>a")
m = a
elif b == a:
print("a=b")
m = a
else:
print("b<a")
m = b
print(m)
# Espressione condizionale
print("b>a") if b>a else print("b<=a")
# Due espressioni condizionali a cascata
print("b>a") if b>a else print("b=a") if b==a else print("b<a")
# Altro esempio
print("b>a" if b>a else "b=a" if b==a else "b<a")
```

2.9 Cicli

```
i = 0
while True:
    i += 1
    if i == 4:
        continue
    print(i)
    if i == 6:
        break

#E' possibile mettere un else fuori da un while, in modo che quando il while
#diventa falso (solo in questo caso) funziona come una sorta di if

i = 0
while i < 6:
    print(i)
    i += 1
else:
    print("Fine del ciclo")
```

```
colori = ["rosso", "verde", "blu"]

for x in colori:
    print(x)

for x in "Visione Artificiale":
    print(x)

for i in range(6):
    print(i)
```

2.10 Liste

```
colori = ['Rosso', 'Verde', 'Blu'] #in alternativa uso il costruttore list()
print(colori)
print(type(colori), len(colori))
print(colori[0]) # Primo elemento
print(colori[-2]) # Penultimo elemento

for c in colori:
    print(c)

colori[1] = 'Grigio' # Modifica di un elemento

colori.append('Verde')
colori.remove('Grigio')

#fa l'append ma con una lista, di solito si utilizza l'operatore +=, che ci da
#ugualmente la possibilita' di concatenare liste

colori.extend(['Arancione', 'Giallo', 'Viola', 'Azzurro'])
print(colori)

#metodologia con +=

l1 = [1, 2]

#mi permette di creare la lista [1, 2, 3, 4], l'id della lista rimane invariato,
#perche' la lista e' un oggetto modificabile, quindi non occorre fare puntare
#la lista nuova ad un nuovo oggetto

l1 += [3, 4]

# Esempi di "list comprehension", modo per generare delle liste con la sintassi
# fra quadre

coloriA = [c for c in colori if c[0]=='A'] #for = tali che
quadrati = [x*x for x in range(1,10)] #lista di tutti i quadrati degli x che
# stanno nel range fra 1 e 9
print(coloriA, quadrati)
```

Accesso agli elementi :

- indice fra parentesi quadre
- Possibile usare indice negativo e intervallo di indici
- iterazione con for..in

L'operazione * con le liste ripete la stessa lista n volte.

2.11 Tuple

Le tuple non sono modificabili, possono essere create anche attraverso il costruttore tuple().

```
colori = ('Rosso', 'Verde', 'Blu')
print(colori)
print(type(colori), len(colori))
print(colori[0]) # Primo elemento
print(colori[-2]) # Penultimo elemento

for c in colori:
    print(c)
t0 = (3) # N.B. questa non e' una tupla, e' un int
t0 = () #questa e' invece una tupla
print(type(t0))

t1 = (3,) # Tupla con un solo elemento
t2 = (1,2)
t3 = (4,5,6)
t = t2 + t1 + t3 # Concatenazione di tuple
tm = t2 * 3 # Moltiplicazione di una tupla
print(t, tm)

# Le parentesi si possono omettere
t1 = 3,
t2 = 1,2 # Oppure 1,2,
t3 = 4,5,6 # Oppure 4,5,6,
print(t1,t2,t3)
```

2.12 Slicing

Informazioni:

- $a[i:j]$ restituisce tutti gli elementi di a con indice k tale che $i \leq k < j$;
- $a[:]$ seleziona tutto;
- l'espressione estesa : $a[i:j:s]$ comprende lo step, ovvero dice quanti elementi saltare all'interno del range indicato. Posso avere anche uno step negativo, quindi restituisco una stringa ribaltata.

```
a = "Python!"  
print(a[4:6], a[0:2], a[:2]) # on Py Py  
print(a[4:len(a)], a[4:100], a[4:]) # on! on! on!  
print(a[:-3], a[-3:]) # Pyth on!  
print(a[1:-1], a[:]) # ython Python!  
print(a[::-2]) # Pto!  
print(a[1::2]) # yhn  
print(a[::-1]) # !nohtyP  
print(a[-3::-2]) # o!  
print(a[-1:-4:-2]) # !o  
a = [1, 2, 3, 4, 5, 6, 7, 8, 9]  
print(a[-2::-2]) # [8, 6, 4, 2]  
#Qui notiamo che e' possibile sfruttare anche l'assegnamento, al posto dei primi  
    3 elementi metto gli altri che ho selezionato  
a[:3] = a[3:]  
print(a) # [4, 5, 6, 7, 8, 9, 4, 5, 6, 7, 8, 9]  
a[::-2] = [0]*6 #qui sostituisco tutta la lista, a step 2, con 6 zeri  
print(a) # [0, 5, 0, 7, 0, 9, 0, 5, 0, 7, 0, 9]  
a[3:-3] = []  
print(a) # [0, 5, 0, 7, 0, 9]  
a[:] = a[::-1]  
print(a) # [9, 0, 7, 0, 5, 0]
```

2.13 Insiemi

Collezione non ordinata e non indicizzata, non può contenere duplicati ma possono essere aggiunti o rimossi elementi (non modificati).

```
colori = {'Rosso', 'Verde', 'Blu'}
print(colori)
print(type(colori), len(colori))
print('Rosso' in colori)

for c in colori:
    print(c)

#Differenza fra remove e discard : remove da' errore se l'elemento non esiste,
#discard no, non da' nessun errore
colori.remove('Verde')
colori.discard('Giallo') # nessun errore
colori.add('Azzurro')
colori.update({'Arancione', 'Viola'})
print(colori)

# Esempi di "set comprehension", stessa sintassi usata con le liste ma con le
# graffe
colori_a = {c for c in colori if c[0]=='A'}
quadrati = {x**2 for x in range(1,10)}
print(colori_a, quadrati)
```

2.14 Dizionari

Non capita spesso in questo corso.

```
studenti = {101:"C.Rossi", 103:"M.Bianchi",111:"L.Verdi"}
print(studenti)
print(type(studenti), len(studenti))
if 103 in studenti:
    print(studenti[103])
studenti[112] = "L.Neri"
studenti[115] = "A.Rosa"
for mat in studenti:
    print(mat, studenti[mat])
for mat, nome in studenti.items(): #items() ritorna chiave-valore
    print(mat, nome)
for nome in studenti.values(): #values ritorna solo i valori
    print(nome)
rimosso = studenti.pop(103)
studenti[103] = rimosso
# Esempi di "dictionary comprehension"
studenti_105 = {m: studenti[m] for m in studenti if m<105}
somme = {(k, v): k+v for k in range(4) for v in range(4)}
print(studenti_105, somme)
```

2.15 Funzioni

I parametri sono riferimenti ad oggetti. Per gli oggetti immutabili, l'effetto è sostanzialmente analogo al passaggio per valore. Per quelli mutabili una funzione può modificare il contenuto di un oggetto.

```
def stampa_messaggio(): # Definizione funzione
    print("Addio, e grazie per tutto il pesce!")
stampa_messaggio() # Chiamata della funzione

def calcola(x, y): # Funzione con parametri
    """Questa e' la docstring di calcola"""
    return x * y
print(f"Il risultato e' {calcola(6, 7)}.")

# Funzione che sostituisce elementi di una lista, enumerate restituisce un
# elenco di tuple con indice e valore
def sostituisci(lista, x, y):
    for (indice, valore) in enumerate(lista):
        if valore == x:
            lista[indice] = y

# Versione piu' "pythonic" della stessa funzione
def sostituisci2(lista, x, y):
    lista[:] = [y if v==x else v for v in lista]

l = [1, 2, 3, 1, 2, 3]
sostituisci(l, 2, 0)
sostituisci2(l, 3, -1)
print(l)
```

2.16 Parametri delle funzioni

```
def calcola(x, y, z = 1, k = 0):
    return (x * y) / z + k
print(calcola(2,3), calcola(2,3,3), calcola(2,3,3,-2))
print(calcola(y=1,x=3), calcola(2,3,k=2),calcola(k=1,x=2,y=3,z=1))

def prodotto(x, *altri_fattori):
    p = x
    for f in altri_fattori:
        p *= f
    return p
print(prodotto(2), prodotto(6,7), prodotto(2,2,2,2,2))

#* = insieme di parametri senza nome (in questo caso gli argomenti
# dell'ipotetico corso
#** = dizionario di parametri con nome
def Esame(corso, *argomenti, **studenti):
    print("Corso:", corso)
    print("Argomenti:", end=' ')
```

```
for a in argomenti:  
    print(a, end=', ')  
    print("\nStudenti:")  
for mat in studenti:  
    print(mat, studenti[mat])  
Esame("Visione Artificiale", " Python", "NumPy", "OpenCV", M101="C.Rossi",  
      M103="M.Bianchi", M111="L.Verdi")
```

2.17 Unpacking

Lo useremo a volte nel passaggio dei parametri.

```
def calcola(a, b, c):  
    return a + b + c  
  
parametri = [4, 18, 20]  
print(calcola(*parametri)) #spacchetta la lista, questo mi permette di passare  
    tre valori separati come argomento della funzione  
  
a = ["Python", "NumPy", "OpenCV"]  
s = {"M101": "C.Rossi",  
      "M103": "M.Bianchi",  
      "M111": "L.Verdi"}  
Esame("Visione Artificiale", *a, **s)  
  
x, y, z = 2, 3, 4 # (x, y, z) = (2, 3, 4)  
  
a1, a2, a3 = a  
print(a1, a2, a3)  
a1, *r = a #questo serve per spacchettare una lista della quale non ne conosco  
    le dimensioni  
print(a1, r)  
primo, secondo, *altri, ultimo = range(10)  
print(primo, secondo, ultimo, altri)  
  
x, y, *_ = a #questo vuol dire che a deve contenere almeno 2 elementi, il primo  
    va in x, il secondo va in y e il resto non mi interessa
```

2.18 Funzioni come oggetti e lambda

```
def prodotto(x, y):
    """Restituisce il prodotto di x e y."""
    return x * y

print(type(prodotto)) # <class 'function'>
print(prodotto.__name__) # prodotto
print(prodotto.__doc__) # la docstring

def esegui(f, x, y):
    """Esegue la funzione f su x e y."""
    return f(x, y)

print(esegui(prodotto, 2, 3))

f = lambda x, y: x ** y
print(type(f)) # <class 'function'>
print(f.__name__) # <lambda>
print(f.__doc__) # None
print(esegui(f, 4, 2))
print(esegui(lambda x, y: x // y, 9, 2))
```

2.19 Funzioni predefinite

```
s = "Python"
a = enumerate(s)
print(list(a))
# [(0,'P'), (1,'y'), (2,'t'),
(3,'h'), (4,'o'), (5,'n')]

n = [ord(c) for c in s]
print(list(n), sum(n), min(n), max(n))# [80, 121, 116, 104, 111, 110] 642 80 121

print(sorted(n))
# [80, 104, 110, 111, 116, 121]

z = zip(s, n)
print(list(z))
# [('P',80), ('y',121), ('t',116),
('h',104), ('o',111), ('n',110)]
```

2.20 Moduli python

```
# Modulo fib.py
def fibonacci(n): # serie di Fibonacci fino a n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result

# Esempi di importazione
import fib
print(fib.fibonacci(90))

import fib as f
print(f.fibonacci(90))

from fib import fibonacci
print(fibonacci(90))

from fib import fibonacci as f
print(f(90))
```

Capitolo 3

NumPy

3.1 Introduzione

Numpy è un modulo fondamentale per il calcolo scientifico, permette di lavorare in maniera molto efficiente su una mole di dati, ad esempio di tipo vettoriale, molto grande. Le strutture dati python sono molto interessanti, però per effettuare calcoli complessi è consigliato sfruttare numPy, dal momento che è scritto in c, c++.

NumPy arricchisce python con l'array-multidimensionale e aggiunge utili funzioni matematiche di base.

3.2 Vantaggi

- le funzioni agiscono a livello vettoriale, questo è vantaggioso, in quanto poche volte si ricorre all'utilizzo di loop esplicativi (che sono più lenti);
- codice scritto in c, c++, dunque è molto efficiente;
- gli algoritmi sono ben testati e progettati (perchè la libreria è molto utilizzata);
- mentre in python le strutture dati possono contenere oggetti qualsiasi, in numPy non è possibile;

3.3 OpenCV e numPy

Numpy è fondamentale perchè fa da wrapper per OpenCV, permette di sfruttarne tutte le possibilità. Il trasferimento di dati in OpenCV avviene grazie a numPy in maniera molto efficiente.

3.4 La classe ndarray

Tutti gli array numPy sono oggetti di questo tipo, implementa un array multidimensionale omogeneo (tutti gli elementi hanno lo stesso tipo). Gli attributi importanti sono:

- ndim: Il numero di dimensioni;
- shape: tupla che indica il numero di elementi lungo ciascuna dimensione
- size: numero totale di elementi dell'array

- dtype: tipo degli elementi
 - itemsize: dimensione in byte di ogni elemento
 - data: il buffer contenente gli elementi (normalmente non serve: si accede agli elementi con le parentesi quadre)
-

```
#Da una lista di liste python il costruttore numpy.array capisce che noi vogliamo creare un array multidimensionale, a conterra' un riferimento all'oggetto numpy creato

a = np.array([[ 0,  1,  1,  2,  3],
              [ 5,  8, 13, 21, 34],
              [ 55, 89, 144, 233, 377]])

print('type:', type(a)) #type:
print('ndim:', a.ndim) #ndim: 2
print('shape:', a.shape) #(3, 5)
print('size:', a.size) #15
print('dtype:', a.dtype) #int32
print('itemsize:', a.itemsize)#4
```

3.5 Creare un array numpy

3.5.1 la funzione array

```
# La funzione array consente di creare un array partendo da una lista o tupla Python
# Il tipo dell'array viene dedotto dagli elementi, oppure puo' essere specificato
a = np.array([2,3,5])
print(a.ndim, a.shape, a.dtype, a.itemsize) #1 (3,) int32 4

a = np.array([2,3,5], np.uint8)
print(a.ndim, a.shape, a.dtype, a.itemsize) #1 (3,) uint8 1

# Sequenze di sequenze sono trasformate in array bidimensionali,
# sequenze di sequenze di sequenze in array tridimensionali e cosi' via
a = np.array([[ 0,  1,  1,  2,  3],
              [ 5,  8, 13, 21, 34],
              [ 55, 89, 144, 233, 377]])
print(a.ndim, a.shape, a.dtype, a.itemsize) #2 (3,5) int32 4
a = np.array([ [ [1,2] ], [ [3,4] ] ]) #lista di liste di liste
print(a.ndim, a.shape, a.dtype, a.itemsize)
#3 (2, 1, 2) int32 4
```

3.5.2 Altri modi per costruire un array

```
# empty() crea un array lasciando i valori non inizializzati, come si puo'
    osservare, attraverso una tupla impongo la dimensione del mio array, posso
    eventualmente aggiungere, come secondo argomento, il tipo di dato.
a = np.empty((2,7), np.int16) # 2 righe, 7 colonne
print(a)

# zeros() e ones(): valori a zero/uno
print(np.zeros(5)) #posso passargli un singolo valore o una tupla, per indicare
    la dimensione
print(np.ones(3, np.int))
print(np.zeros((2,3)))

#arange() e' simile a range() di Python
a = np.arange(100, 110, 2)
print(a)
#identity() crea una matrice identita'
a = np.identity(3)
print(a)

#Output
[[ -4352 26016 546 0 4 0 1]
 [ 29184 -16960 25809 546 0 -16992 25809]]

[0. 0. 0. 0. 0.]
[1 1 1]
[[0. 0. 0.]
 [0. 0. 0.]]
[100 102 104 106 108]

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

3.6 Operazioni di base

```
# Gli operatori aritmetici si applicano agli array elemento-per-elemento:  
# il risultato viene tipicamente memorizzato in un nuovo array  
a = np.array( [25,36,49,64] )  
b = np.arange( 4 ) #specificando solo un valore si intende da 0 fino a 4  
    escluso, funziona come range  
print(b)  
  
#Tutte le operazioni fra array numpy sono eseguite elemento per elemento,  
    infatti, per calcolare c, e' necessario che a e b abbiano la stessa  
    dimensione.  
c = a-b  
print(c)  
  
print(b**2) #eleva a potenza un array numpy, elemento per elemento.  
  
print(np.sqrt(a)) #questo e' il primo esempio di ufunc(), ovvero universal  
    function, posso passargli un array numpy.  
  
# I confronti seguenti producono array di  
# valori booleani con le stesse dimensioni di a  
print(a < 37) #scrivere che una matrice a e' minore di 37, significa confrontare  
    elemento per elemento, verificando che tutti i componenti dell'array siano  
    minori di 37.  
  
b = np.array( [25,37,49,63] )  
print(a == b) #e' un array di booleani, confronta elemento per elemento i due  
    array, e inserisce true se sono uguali e false altrimenti. ATTENZIONE: non  
    sta controllando se a e b sono due array numericamente uguali, per fare cio'  
    occorre usare np.equals(a, b)  
  
. #Output  
  
[0 1 2 3]  
  
[25 35 47 61]  
  
[0 1 4 9]  
  
[5. 6. 7. 8.]  
  
[ True True False False]  
  
[ True False True False]
```

3.7 Operazione prodotto

```
# Anche l'operatore prodotto '*' opera elemento-per-elemento;
# Dalla v3.5 di Python e' disponibile l'operatore '@' per i prodotti fra matrici
A = np.array( [[1,1],
               [0,1]] )
B = np.array( [[2,0],
               [3,4]] )

print(A * B) # Prodotto elemento-per-elemento:(1*2 = 2; 1*0 = 0; 0*3 = 0; 1*4 =4)

print(A @ B) # Prodotto tra matrici (riga * colonna, faccio la prima riga di A *
              # prima colonna di B e cosi' via), molto elegante

# Un vettore "colonna": shape = (3, 1)
c = np.array([[1], [2], [3]])
# Un vettore "riga": shape = (1, 3)
r = np.array([[3,0,2]])

print(c @ r) # (3,1)x(1,3) = (3,3)

print(r @ c) # (1,3)x(3,1) = (1,1)

#Output
[[2 0]
 [0 4]]

[[5 4]
 [3 4]]

[[3 0 2]
 [6 0 4]
 [9 0 6]]

[[9]]
```

3.8 Operatori di assegnamento

```
# Gli operatori di assegnamento come += o *= modificano
# l'array esistente senza doverne creare uno nuovo

a = np.ones((2,3))
b = np.array([[1,0,2], [0,3,0]])

id_a = id(a)

a *= b #viene modificato il contenuto di a senza che venga riallocato un altro
       oggetto, basti ricordare che le liste sono modificabili

print(a)

print(id_a, id(a), id_a==id(a))

a = a * b

print(a)

print(id_a, id(a), id_a==id(a))

#Output

[[1. 0. 2.]
 [0. 3. 0.]]

2346790588544 2346790588544 True

[[1. 0. 4.]
 [0. 9. 0.]]

2346790588544 2346791113632 False
```

3.9 Type cast

```
a = np.ones(3, np.int32)
b = np.array([1.4, 1.5, 1.6])
print(a.dtype, b.dtype)
# In una operazione fra array di tipo diverso,
# il tipo del risultato corrisponde a quello piu'
# preciso (o piu' generale) dei due
c = a + b
print(c)
print(c.dtype)
# Per creare un nuovo array cambiando il tipo di un
# array esistente, si puo' usare il metodo astype()
a = np.arange(10, dtype = np.uint8)
print(a.dtype)
a = a.astype(np.uint64)
print(a.dtype)

#Output

int32 float64
[2.4 2.5 2.6]

float64

uint8

uint64
```

3.10 Alcune operazioni su un array

```
#Crea una matrice contenente numeri casuali nell'intervallo [0,1)

a = np.random.random((2,3))
print(a)

# Calcola il valore minimo, massimo e la somma
print(f'Min: {a.min()}'')
print(f'Max: {a.max()}'')
print(f'Sum: {a.sum()}'')

# Il calcolo, invece che su tutti gli elementi,
# puo' essere lungo uno specifico asse

print(f'Somma di ogni colonna: {a.sum(axis=0)}')
print(f'Somma di ogni riga: {a.sum(axis=1)}')
```

3.11 Funzioni universali (ufunc)

Funzioni numpy che operano su ndarray elemento-per-elemento. Alcuni esempi:

- Operazioni matematiche;
- Trigonometria;
- Operazioni sui bit;
- Confronto;
- Floating point.

3.12 Indicizzazione e slicing su array monodimensionali

```
# Negli array NumPy monodimensionali, l'accesso agli elementi e' analogo a
# quanto avviene con liste e altre sequenze in Python
a = np.arange(10)**3 #10 elementi e ogni singolo elemento lo elevo al cubo
print(a)
print(a[2]) # Accesso a un elemento
print(a[2:5]) # Slicing: dall'elemento 2 al 4
a[:6:2] = 42 # Modifica elementi di posto 0, 2, 4
print(a)
print(a[::-1]) # Step negativo: ordine inverso
a[::2] += a[1::2] # a[i] += a[i+1], i=0,2,4,6,8
print(a)
a[:] = -1 # Modifica tutti gli elementi
print(a)

#Output

[ 0  1  8  27  64  125  216  343  512  729]

8

[ 8 27 64]

[ 42  1  42  27  42  125  216  343  512  729]

[729 512 343 216 125 42 27 42 1 42]

[ 43  1  69  27  167  125  559  343  1241  729]

[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

3.13 Indicizzazione e slicing su array multidimensionali

La funzione `fromfunction` è molto utile, permette di passare una funzione python (che prende come parametri due interi, perchè ci sono due dimensioni) e sulla base della funzione crea un array multidimensionale delle dimensioni definite e con valori ottenuti applicando la funzione.

Indicizzazione e slicing su array multidimensionali

```
a = np.fromfunction(lambda i,j: i*10+j, (3,5), dtype=int)
print(a)

# Riga 2, Colonna 3
print(a[2, 3])

# Righe 0 e 1, colonna 2
print(a[:2, 2])

# La colonna 1
print(a[:, 1])

# La riga 1
print(a[1, :])

# La riga 1: eventuali indici mancanti
# sono sostituiti con ':'
print(a[1])
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]]
```



```
23
```

```
[ 2 12]
```

```
[ 1 11 21]
```

```
[10 11 12 13 14]
```

```
[10 11 12 13 14]
```

Nello slicing di matrici, se ometto il secondo termine, facendo ad esempio `a[1:3]` è come se sottointendessi `a[1:3,:]`, quindi righe da 1 a 3 poi tutte le colonne.

```
a = np.fromfunction(lambda i,j: i*10+j, (3,5), dtype=int)
print(a)

# Righe 0 e 2, colonne 0 e 3
print(a[::2, ::3])

# Righe 1 e 2
print(a[1:3])

# Righe 0 e 1, colonne 0, 1 e 2
print(a[:2, :3])

# Ultime due righe
print(a[-2:])
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]]
```



```
[[ 0  3]
 [20 23]]
```

```
[[10 11 12 13 14]
 [20 21 22 23 24]]
```

```
[[ 0  1  2]
 [10 11 12]]
```

```
[[10 11 12 13 14]
 [20 21 22 23 24]]
```

3.14 Iterare un array

Di solito non si fanno iterazioni usando numpy, si riesce quasi sempre a trasformare dei cicli in funzioni vettoriali. Nonostante cio' puo' capitare di iterare.

```
a = np.arange(7)
print(a)
# Si puo' iterare sugli elementi un array monodimensionale

for x in a:
    print(x, end='; ')
print()

a = np.fromfunction(lambda i,j: i*10+j, (3,5), dtype=int)
print(a)

# Iterando su una matrice si ottengono le righe...
for r in a:
    for x in r: # ... su cui si puo' ulteriormente iterare
        print(x, end='; ')
    print()
# L'attributo .flat e' un iteratore su tutti gli elementi
for x in a.flat:
    print(x, end=', ')

#Output
[ 0 1 8 27 64 125 216 343 512 729]
8
[ 8 27 64]
[ 42 1 42 27 42 125 216 343 512 729]
[729 512 343 216 125 42 27 42 1 42]
[ 43 1 69 27 167 125 559 343 1241 729]
[-1 -1 -1 -1 -1 -1 -1 -1 -1]
```

3.15 Slicing con ellissi

Utile quando ho un array con molti assi (3 dimensioni o più), i puntini ci evitano di dover precisare tutte le dimensioni.

Slicing con ellissi (...)

21

- Ellipsis: oggetto Python predefinito che può essere indicato semplicemente con ...
- NumPy lo utilizza nello slicing per rappresentare tutti i ':' che mancano per produrre una tupla di indicizzazione completa. Ad esempio se v è un array con 5 dimensioni:
 - $v[1, 2, \dots]$ equivale a $v[1, 2, :, :, :]$
 - $v[\dots, 3]$ equivale a $v[:, :, :, :, 3]$
 - $v[4, \dots, 5, :]$ equivale a $v[4, :, :, 5, :]$
- Ovviamente questo è utile solo in array con tre o più dimensioni

```
# Inizializza un array con 3 dimensioni
c = np.array( [[[ 0,  1,  2],
                 [ 10, 12, 13]],
                [[ 50, 51, 52],
                 [ 60, 62, 63]]])
print(c.shape)
print(c[1,...]) # Equivale a c[1] e a c[1,:,:]
print(c[:,...,2]) # Equivale a c[:, :, 2]
```

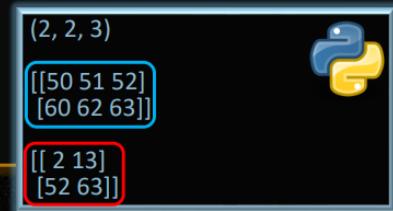
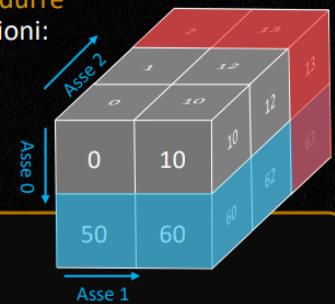


Figura 3.1: Esempio di periodo

3.16 Modifica della forma

Differenza reshape e resize : reshape costruisce un nuovo array e lo restituisce, resize non restituisce nulla, modifica l'array sulla quale viene chiamato il metodo.

```
a = np.arange(12)
print(a.shape)
print(a)

# reshape() restituisce gli stessi con forma diversa
b = a.reshape(2, 6)
print(b.shape)
print(b)

# Se una dimensione e' -1, viene calcolata automaticamente
c = a.reshape(2, 2, -1)
print(c.shape)
print(c)

a.resize(2,6) # resize() modifica l'array stesso
print(a)

# ravel() restituisce i dati come array monodimensionale
d = a.ravel()
print(d)
```

```
#Output

(12,)
[ 0 1 2 3 4 5 6 7 8 9 10 11]
(2, 6)
[[ 0 1 2 3 4 5]
 [ 6 7 8 9 10 11]]
(2, 2, 3)
[[[ 0 1 2]
 [ 3 4 5]]
 [[ 6 7 8]
 [ 9 10 11]]]
[[[ 0 1 2 3 4 5]
 [ 6 7 8 9 10 11]]
 [ 0 1 2 3 4 5 6 7 8 9 10 11]]]
```

3.16.1 Altri modi per modificare la forma

C'è la possibilità di aggiungere nuove dimensioni con cardinalità uno con np.newaxis.

```
a = np.arange(12).reshape(2,6)
print(a)

# Aggiunta di nuove dimensioni con np.newaxis
b = a[np.newaxis, ...] #Aggiungo alla shape di a un nuovo asse, serve se mi
# occorre operare su un array di dimensione diversa rispetto a quella attuale
print(b.shape)
print(b)
c = a[np.newaxis, ..., np.newaxis, np.newaxis]
print(c.shape)

# np.newaxis non e' altro che un riferimento a None
print(np.newaxis is None)

# Scambiare le dimensioni con transpose() o .T
d = a.T
e = a.transpose()
print(d.shape, e.shape)

# squeeze() elimina eventuali dimensioni a uno, torna utile nel caso in cui mi
# serve togliere le dimensioni in piu'.
f = c.squeeze()
print(f.shape)
```

```
#Output

(12,)
[ 0 1 2 3 4 5 6 7 8 9 10 11]
(2, 6)
```

```
[[ 0 1 2 3 4 5]
 [ 6 7 8 9 10 11]]
(2, 2, 3)
[[[ 0 1 2]
 [ 3 4 5]]
 [[ 6 7 8]
 [ 9 10 11]]]
[[ 0 1 2 3 4 5]
 [ 6 7 8 9 10 11]]
[ 0 1 2 3 4 5 6 7 8 9 10 11]
```

3.17 Concatenare array

```
#Qui ha creato una matrice quadrata random di dimensione 2, moltiplicando ogni
elemento per due. Con np.floor poi ha effettuato il troncamento sui numeri.

a = b = np.floor(10*np.random.random((2,2)))
print(a)

print( np.vstack((a,b)) ) #vstack : vertical stack, mette in pila in verticale

print( np.hstack((a,b)) ) #hstack : horizontal stack

# column_stack() affianca array 1D come colonne
# di un array 2D
#Ho due array monodimensionali che vengono considerati vettori colonna e vengono
affiancati, e' evidente la differenza con vstack.
x = np.array([4.,2.])
y = np.array([3.,8.])
print( np.column_stack((x,y)) )
#notare la differenza se invece di usa hstack()
print( np.hstack((x,y)) )
#Output
[[5. 6.]
 [2. 0.]]


[[4. 8.]
 [1. 7.]]


[[5. 6.]
 [2. 0.]
 [4. 8.]
 [1. 7.]]


[[5. 6. 4. 8.]
 [2. 0. 1. 7.]]


[[4. 3.]
 [2. 8.]]


[4. 2. 3. 8.]
```

3.18 Copie e viste di array

Tante operazioni in numpy non producono un nuovo array con una copia dei dati, producono un nuovo oggetto ma i dati che contiene questo oggetto array sono gli stessi. Noi sappiamo che l'assegnamento non crea un nuovo oggetto ma copia solamente il riferimento. Nemmeno lo slicing non crea nuovi oggetti, è un concetto chiave di python, cambia i riferimenti.

```
a = np.arange(7)
print(a)

# Come per qualsiasi variabile Python, l'assegnamento
# non crea un nuovo oggetto, solo un nuovo riferimento
b = a
print(b is a)

# Lo slicing (come altre operazioni) crea una vista: un
# nuovo oggetto che condivide gli stessi dati. a.base
# e' un riferimento all'oggetto su cui e' costruita la vista
c = a[3::2]
print(c is a, c.base is a) #L'attributo base di un oggetto numpy e' None se
    # contiene i propri dati e non e' la lista di un altro oggetto, se invece e'
    # diverso da None e' una lista che contiene i propri dati. Numpy tenta di
    # eliminare le duplicazioni inutili.
print(c)
c[0] = -1
print(a) #Da questa print capiamo che sono oggetti diversi ma i dati sono gli
    # stessi! Modificando i dati di c vengono di conseguenza modificati quelli di
    # a. Quindi occorre ricordare che quando estraiamo gli elementi con lo
    # slicing, in realta' otteniamo una lista sugli stessi dati... ma come faccio
    # ad evitarlo? con il metodo copy()

d = a.copy() # Il metodo copy() crea una copia dell'array
print(d is a, d.base is a, d.base)
d[0] = -1
print(d, a)
```

#Output

[0 1 2 3 4 5 6]

True

False True

[3 5]

[0 1 2 -1 4 5 6]

False False None

[-1 1 2 -1 4 5 6] [0 1 2 -1 4 5 6]

3.19 Broadcasting

- Comodo e potente strumento con cui NumPy consente alle ufunc di agire su input con forme diverse.
 - Prima regola: se gli array di input non hanno lo stesso numero di dimensioni, un "1" viene anteposto alla shape dell'array più piccolo finché il numero di dimensioni non coincide.
 - Seconda regola: le dimensioni a 1 sono trattate come se il numero di elementi fosse pari a quello dell'array con più elementi lungo le corrispondenti dimensioni: si assume che tutti gli elementi lungo tale dimensione siano uguali.
 - Se dopo l'applicazione di queste due regole gli array hanno la stessa forma, l'operazione viene eseguita.

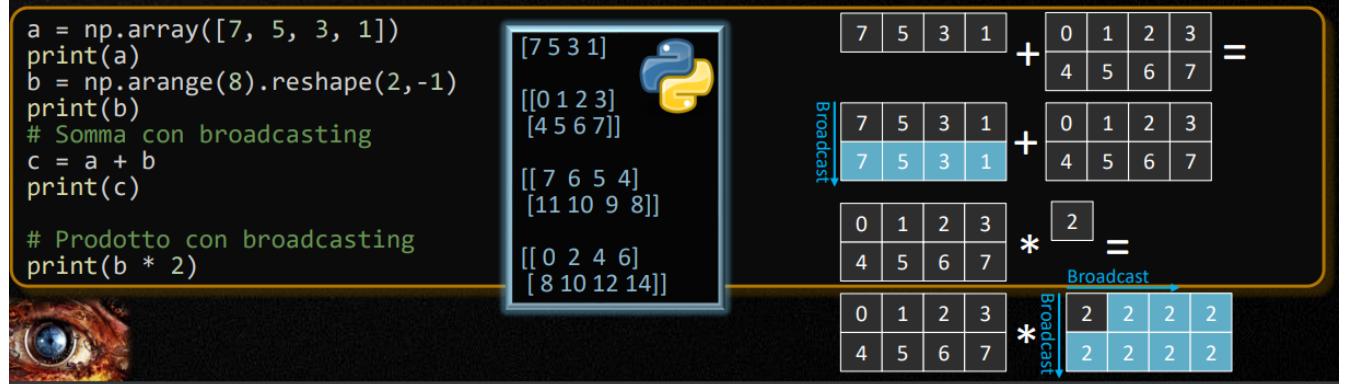


Figura 3.2: Esempio di periodo

3.20 Indicizzare con un array di indici

```

a = np.arange(12)**2
print(a)
# Oltre a interi e slicing, si possono
# utilizzare array di interi per
# indicizzare altri array
idx1 = np.array( [ 1,1,3,8,5 ] )
print(a[idx1])
# L'array di interi puo' anche
# essere multidimensionale: il
# risultato ha la forma dell'array
# usato come indice
idx2 = np.array( [ [3, 4], [9, 7] ] )
print(a[idx2])

```

#Output

[0 1 4 9 16 25 36 49 64 81 100 121]

[1 1 9 64 25]

[[9 16]
[81 49]]

3.21 Indicizzare un array multidimensionale con un array di indici

```
# Se l'array a e' multidimensionale si puo' passare un array
# di indici per ciascuna dimensione, purché abbiamo la stessa
# forma (o si possa fare broadcast fra loro).
# Come per lo slicing, si puo' utilizzare ':' o '...'
# per non dover specificare tutte le dimensioni.
a = (np.arange(12)**2).reshape(3,4)
print(a)

idx2 = np.array([1,1,2])
print(a[idx2,:]) # oppure print(a[idx2])
print(a[:,idx2])

idx_r = np.array( [ [0,0,0], [1,1,1] ] )
idx_c = np.array( [ [2,3,2], [0,0,0] ] )
print(a[idx_r,idx_c])

#Output

[[ 0  1  4  9]
 [ 16 25 36 49]
 [ 64 81 100 121]]

[[ 16 25 36 49]
 [ 16 25 36 49]
 [ 64 81 100 121]]

[[ 1 1 4]
 [ 25 25 36]
 [ 81 81 100]]

[[ 4 9 4]
 [16 16 16]]
```

3.22 Indicizzare con array di booleani

```
# Nell'indicizzazione con array di interi si specificano
# gli indici da considerare, invece in questo caso si scelgono
# esplicitamente quali elementi si vogliono e quali no.
# Il modo piu' semplice e' utilizzare array booleani con la
# stessa forma dell'array originale.
a = np.arange(12).reshape(3,4)
print(a)

b = a > 4
print(b) # array booleano con la stessa forma di a

print(a[b]) # un array 1D con gli elementi considerati

# La selezione con elementi booleani e' molto utile per
# modificare solo gli elementi che soddisfano un certo criterio.
criterio = a%3 == 0
print(criterio)

print(a[criterio])

#Output
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

[[False False False False]
 [False True True True]
 [ True True True True]]

[ 5  6  7  8  9 10 11]

[[ True False False True]
 [False False True False]
 [False True False False]]

[0  3  6  9]
```

Capitolo 4

Immagini

4.1 Le immagini digitali

Una immagine digitale è definita da un insieme di pixel, ogni pixel rappresenta una informazione che viene campionata e quantizzata (ad esempio il colore è quantizzato, in quanto assume un numero finito di valori) misurata da un sensore. Nota : quantizzare significa quantificare. Lavoreremo molto con le immagini grayscale.

4.1.1 Caratteristiche di una immagine

- Dimensione (numero di pixel in larghezza = WxH) e risoluzione(DPI);
- formato dei pixel (possono essere o bianchi o neri oppure a colori)
- formati di memorizzazione e compressione;
- occupazione di memoria di una immagine (si misura in funzione dell'ampiezza per altezza per profondità, in sigla si indica come : WxHxDepth)

4.1.2 Immagine greyscale con punti di luce

Vediamo una immagine come una matrice di punti di luce (ogni elemento della matrice rispetta una scala di luminosità). Ogni pixel della matrice assume un valore, il quale indica la quantità di luce.

- valori più alti : maggiore luminosità
- valori più bassi : minore luminosità
- intervalli tipici : Byte[0, 255] e Float[0 = nero, 1 = luce]

4.1.3 Coordinate dei pixel

Ci sono due tipi di coordinate:

- espresse in forma cartesiana (`cv.line(img, (4, 0), (3, 6), color))`;
- espresse in notazione matriciale (`img[3, 1] = x`).

4.1.4 Organizzazione dei pixel in memoria

I pixel sono quasi sempre organizzati in memoria per righe, ricordando che la memoria dell'elaboratore non è una matrice, bensì un unico vettore unidimensionale di byte.

4.2 OpenCV

Le immagini le definisco come array numpy bidimensionali. Attenzione imread non ritorna un errore o una eccezione nel caso in cui l'immagine non esista, ritorna un None, occorre dunque fare un controllo.

```
# creazione di un'immagine grayscale 16x15 con un byte per pixel

img1 = np.zeros((15, 16), dtype=np.uint8) # Tutti i pixel a 0
img2 = np.full((15, 16), 255, dtype=np.uint8) # Tutti i pixel a 255
img3 = np.random.randint(0, 256, (15, 16), dtype=np.uint8) # Valori dei pixel casuali

# Caricamento di un'immagine grayscale da file (se e' a colori viene convertita)
# N.B. se il file non esiste non genera un errore ma restituisce None

#il flag indicato ci dice che anche se l'immagine fosse a colori verrebbe convertita in greyscale
img4 = cv.imread('esempi/mario.png', cv.IMREAD_GRAYSCALE)
```

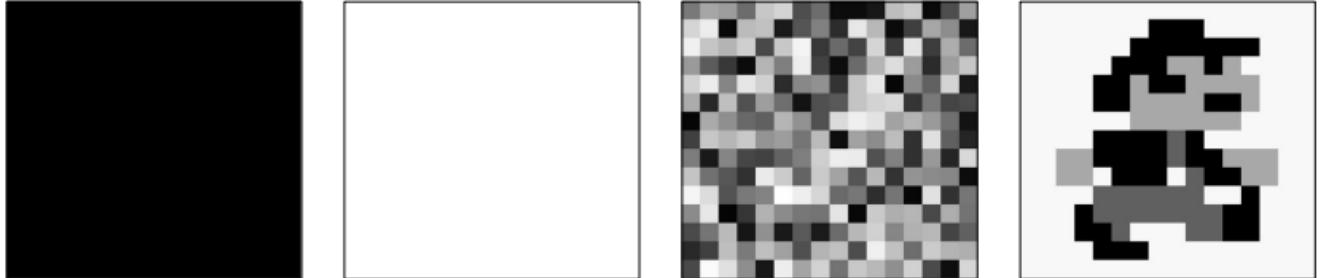


Figura 4.1: Immagini generate con opencv

In seguito viene mostrato come utilizzare slicing e broadcasting per modificare una immagine in python.

```
a = np.full((15, 16), 255, dtype=np.uint8) #immagine tutta bianca
# Slicing e broadcasting
#partendo dalla terza riga fino alla penultima esclusa, viene fatta la stessa
    cosa per le colonne
a[2:-2, 2:-2] = 0 # creo una parte nera, a partire dalla riga di indice 2
a[4:-4, 4:-4] = 128 # creo una parte grigio chiaro
a[6:-6, 6:-6] = 64 # grigio con altra tonalita'

b = a.copy()
# Boolean indexing e broadcasting
mask = b==64 # creo una maschera di bool che conterrà true per tutti i valori
    di b uguali a 64, dunque sostituisco la parte centrale con il bianco
b[mask] = 240 #questo array viene esteso attraverso il broadcasting

c = np.zeros_like(a) #zeros_like() prende un array e restituisce un array con lo
    stesso tipo e con le stesse caratteristiche ma tutto a zero
# Slicing e broadcasting
c[:,::2, ::3] = 255

d = np.zeros((15, 16), dtype=np.uint8) #immagine nera
# Integer array indexing e broadcasting
y = np.array([0, 2, 5, 9, 14]) #array di indici monodimensionale, seleziona le
    righe sulla quale voglio lavorare
x = np.arange(0, 16, 2) #seleziona le colonne che voglio modificare
#qui sto utilizzando questi array come indici
d[y[:, np.newaxis], x] = np.arange(50, 255, 50)[:,np.newaxis]
```

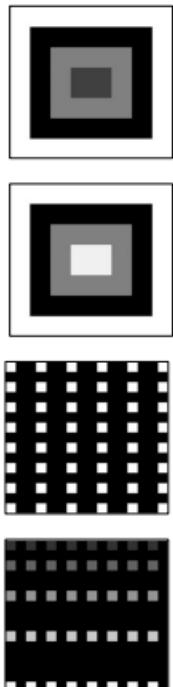


Figura 4.2: Immagini generate con opencv

4.3 Immagini a colori: tensori 3D

Le immagini a colori si modellano mettendo insieme componenti di base (rgb). Una immagine a colori è come se fosse la sovrapposizione di 3 immagini con una determinata scala di valori. Quindi abbiamo che le immagini a colori sono definite da array tridimensionali, la profondità è data dai canali. Sovrapponendosi fra loro dunque, i canali conferiscono il colore all'immagine.

4.3.1 Modello RGB

Ci sono tanti modelli, in questo corso vediamo il minimo indispensabile per capire di cosa si tratti. Il modello di base è l'RGB, è utilizzato ad esempio per generare colore nei monitor, è un modello additivo dal momento che combinando il verde il blu e il rosso si ottiene tutta la varietà di colori. Alla fine abbiamo questi 3 colori ed ogni colore lo possiamo immaginare come un punto in uno spazio tridimensionale (asse del rosso, asse del verde e asse del blu). Per effettuare operazioni di riconoscimento degli oggetti (somiglianza fra colori), lo spazio euclideo del cubo RGB non va bene, perchè i colori che sono simili non sempre sono vicini.

4.3.2 Coordinate dei pixel

Notazione:

- cartesiane : $\text{pixelpos} = (14, 3, 1)$, sto indicando x, y e canale;
- matriciale : $\text{img}[14, 4, 0]$, sto indicando r, c, canale.

4.3.3 Ordine dei canali

L'ordine dei canali sfruttato normalmente è RGB ma attenzione! Open CV utilizza il modello BGR ($B = 0$, $G = 1$, $R = 2$). In caso di cambio ordine occorre effettuare delle modifiche.

4.3.4 Quale è l'ordine dei pixel in memoria?

Quasi sempre viene utilizzato un canale in modo contiguo.

4.4 Caricamento di immagini a colori

```
# Creazione di un'immagine 16x20 a 3 canali con valori di tipo byte (3 byte per pixel)

img1 = np.zeros((20, 16, 3), dtype=np.uint8) # Pixel a 0
img2 = np.full((20, 16, 3), 255, dtype=np.uint8) # Pixel a 255
img3 = np.random.randint(0, 256, (20, 16, 3), dtype=np.uint8) # Pixel casuali

# Caricamento di un'immagine da file (formato BGR)
# N.B.: se il file non esiste non genera un errore ma restituisce None
img4 = cv.imread('esempi/mario-c.png')

# Caricamento di un'immagine da file (formato BGR)
m = cv.imread('esempi/mario-c.png')

# Crea tre immagini BGR ciascuna con valori solo in un canale e gli altri due a zero
b, g, r = m.copy(), m.copy(), m.copy()
#Qui voglio azzerare tutti i canali tranne 1, in modo da ottenere delle immagini monocromate. Lo faccio sfruttando l'ellipsis notation e lo slicing.
b[...,1:3], g[...,0:3:2], r[...,0:2] = 0, 0, 0

# Crea tre immagini BGR ciascuna corrispondente alla somma di due canali
c, m, y = b+g, b+r, g+r
```



Figura 4.3: Prime immagini a colori



Figura 4.4: Altre immagini a colori

4.5 Array numpy tridimensionali

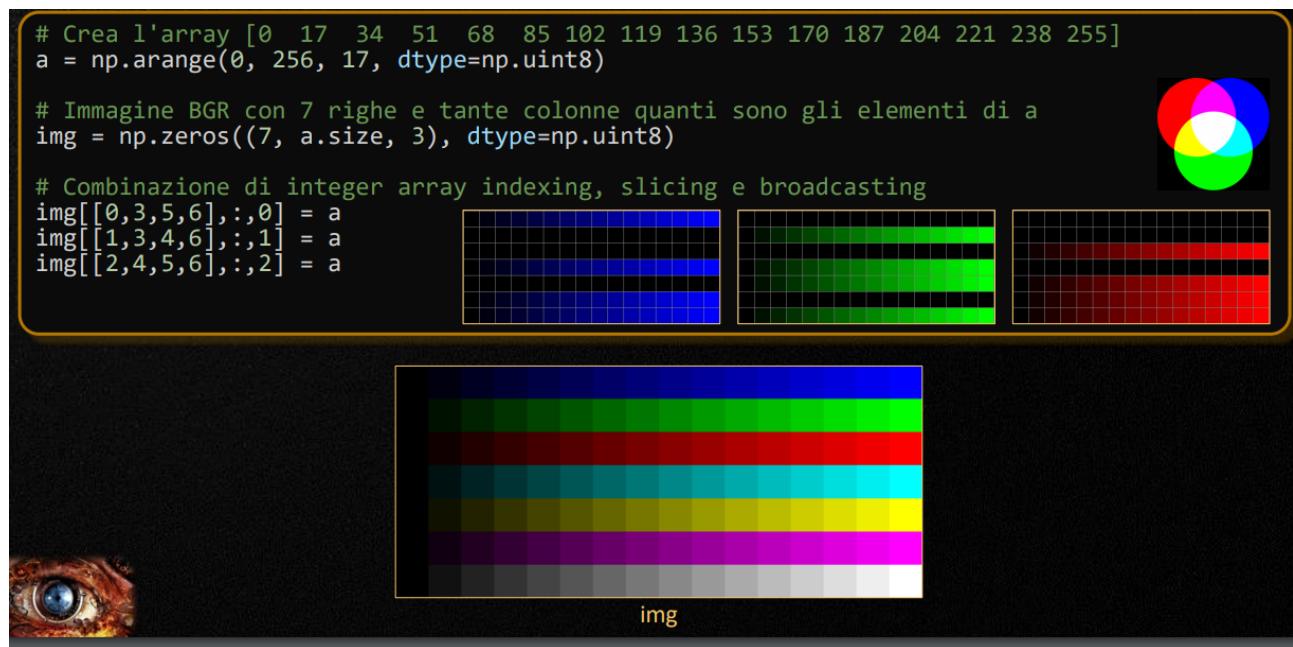


Figura 4.5: Indicizzazione di array di interi, nell'ultima parte sto dicendo : nel canale 0, su tutte le colonne e su una serie di righe, copio la configurazione definita nel canale a. Faccio la stessa cosa sul canale 1 e 2.

```
t = cv.imread('immagini/toys.png')

f = (t[::-1], t[:,::-1], t[::-1,:,:-1], t[...,:-1])

h, w = t.shape[:2]
h2, w2 = h//2, w//2

s = (t[:, :w2], t[:, :h2], t[:, :h2, :w2], t[:, :h2, w2:], t[:, h2:, :w2], t[:, h2:, w2:])

r = [t[:, ::k, ::k] for k in range(2, 30, 2)]
```

Figura 4.6: Qui utilizziamo lo slicing per fare a fette le immagini. Nel primo caso sto generando una tupla di array numpy

4.6 Istruzione shape

```
t = cv.imread('path')

t.shape #mi mostra la shape dell'immagine : (righe, colonne, canali)

#un amateur fa cosi'...

w = t.shape[1] #numero delle righe
h = t.shape[0] #numero di colonne

print("Dimensione immagine", w, x, h)

#un super python expert giga chad fa invece cosi'!

h, w, c = t.shape #questo perche' conosco lo scompattamento delle tuple

h, w = t.shape #faccio cosi' se il valore del canale non mi interessa
```

4.7 Rappresentazioni HSV HSL

Introduciamo due modelli di colori più comodi per noi esseri umani. Sono basate su:

- Hue(tinta) : è un angolo;
- Saturation(saturazione) : quanto il colore è bello vivo, va dai colori spenti ai più accesi;
- Value o lightness: dice quanta luce c'è nel colore, va dal nero al bianco.

4.7.1 Differenze HSV e HSL

In HSV i colori più saturi hanno luminosità 1, mentre nell'HSL 0.5. Ci sono altre caratteristiche che non verranno affrontate nello specifico in questo corso.

4.7.2 Modifica ai canali HSL

Creo un mapping, sfruttando la scala greyscale, fra tutti i possibili angoli (che corrispondono ai colori), consideriamo i valori fra 0 a 255. Per quanto riguarda la saturazione, nel mapping greyscale, si vede in base all'oggetto più luminoso (colore più spento = colore poco saturo, colore più acceso = colore molto saturo). Alla fine della fiera, in questa immagine viene mostrato come determinare i valori H, S ed L attraverso un intelligente mapping della scala greyscale. Ad esempio, se eseguo una somma sullo HUE sto in realtà ruotando la ruota dei colori, per quello utilizziamo le notazioni in radianti.

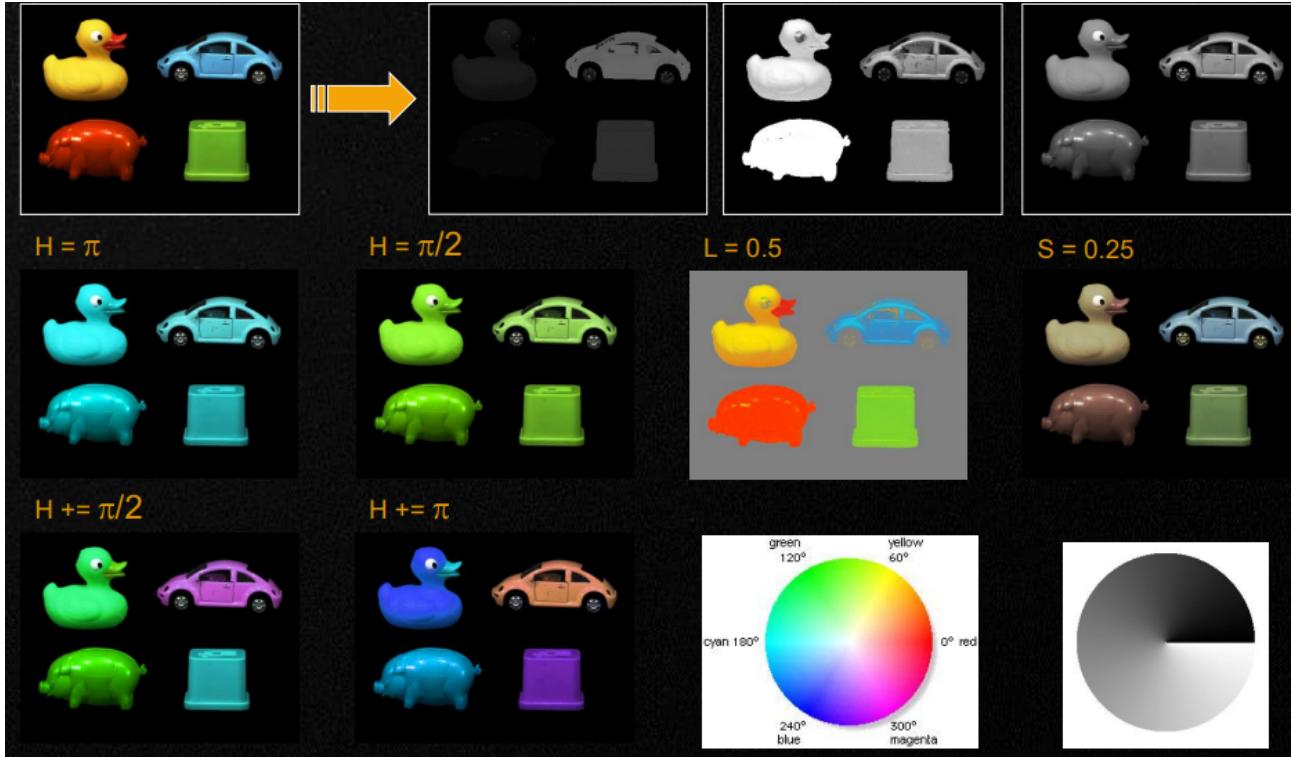


Figura 4.7: Qui utilizziamo lo slicing per fare a fette le immagini. Nel primo caso sto generando una tupla di array numpy

4.7.3 HSV e HSL in python/openCV

La funzione cv.cvtColor(originale, CONV) effettua la conversione del formato del colore, lo si fa specificando una costante come secondo argomento. Attenzione! In python invece di HLS viene considerato HSL.

```

originale = cv.imread('immagini/toys.png')
# Converte in HLS
hls = cv.cvtColor(originale, cv.COLOR_BGR2HLS)
# Separa i tre canali in 3 immagini grayscale
h, l, s = cv.split(hls)

# Modifica alcuni dei valori HLS
# Dimezza la luminosita' di tutti i pixel

l1 = l//2

# Valore di saturazione 64 per tutti i pixel

```

```

s1 = np.full_like(s, 64)

# Riunisce i canali e converte in BGR
#merge e' il corrispettivo di merge, vuole un solo parametro e deve essere una
# tupla

risultato = cv.cvtColor(cv.merge((h, l1, s1))

```



```

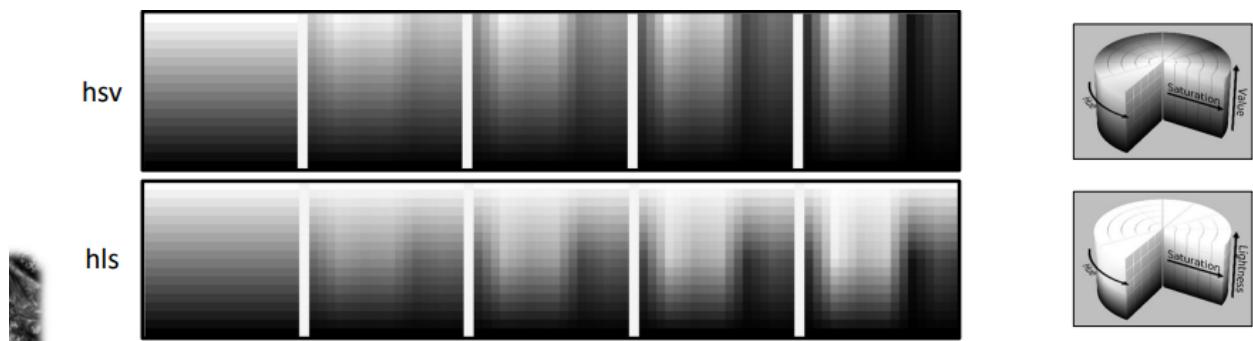
s = [0, 64, 128, 192, 255] # 5 livelli di saturazione

# 18 valori di luminosita' in ogni colonna, np.tile permette di replicare tutte
# le volte che voglio un array, creando un array piu' grande
v = np.tile(np.arange(255, -1, -15, np.uint8)[:,np.newaxis], (1, 18))

# 18 valori di hue in ogni riga
h = np.tile(np.arange(0, 180, 10, np.uint8), (18, 1))

# Combina i 3 canali (si poteva usare anche cv.merge)
hsv = [np.dstack((h, np.full_like(h, x), v)) for x in s]
hls = [i[...,[0,2,1]] for i in hsv] # Scambia gli ultimi due canali

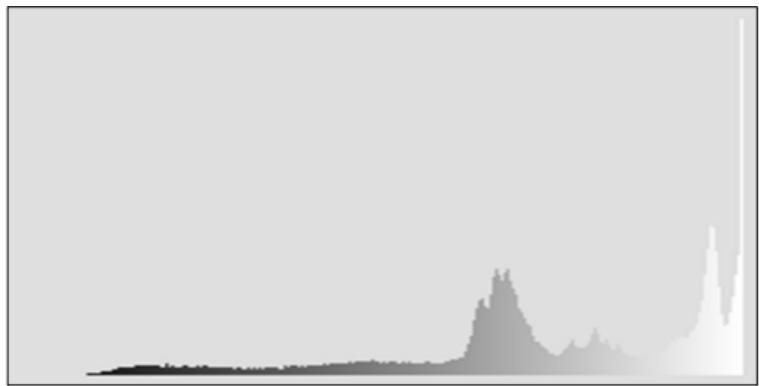
```



4.8 Istogramma di una immagine grayscale

Un istogramma è un grafico che ci dice quanti pixel dentro all'immagine ci sono per ciascuno dei possibili livelli di grigio. Nell'esempio abbiamo a sinistra una immagine con 1 byte per ogni pixel, quindi ogni immagine può avere valori da 0 a 255. Il nostro istogramma avrà 255 possibili colonne, da 0 a 255 e l'altezza di ogni colonna di questo istogramma sarà uguale a quanti pixel di quel livello di grigio ci sono in tutto nell'immagine. L'istogramma nelle immagini è importante, perchè permette di estrarre informazioni interessanti, quali:

- se la maggior parte dei valori sono solo condensati in una zona, l'immagine ha uno scarso contrasto;
- se nell'istogramma sono predominanti le basse intensità, l'immagine è molto scura.



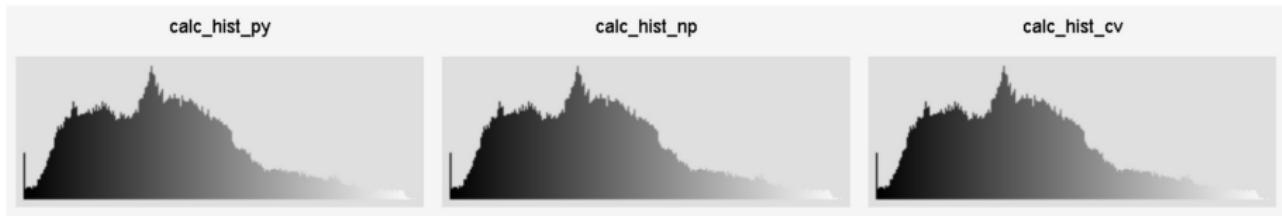
4.9 Calcolo istogramma

```
#da non fare in un progetto reale, perche' e' lentissimo, il python non e'
#adatto ad eseguire dei cicli
def calc_hist_py(img):
    h = np.zeros(256, dtype=int)
    #np.nditer e' un iteratore
    for p in np.nditer(img):
        h[p] += 1 #questo mi alza la colonnina dell'indice p
    return h

#np.histogram(immagine, numero di elementi dell'istogramma, range dei valori) e'
#generale, non specifico per le immagini. Questo metodo restituisce una tupla
#con 2 valori, il primo valore e' l'array dell'istogramma, il secondo valore
#della tupla e' l'elenco dei range. Per questo e' stato considerato solo
#l'elemento di indice 0.
def calc_hist_np(img):
    return np.histogram(img, 256, [0,256])[0]

#cv.calcHist() e' specifico per le immagini, calcola tanti istogrammi in un
#colpo solo. Noi gli passiamo (lista python, canale(lista), ). Restituisce un
#array bidimensionale, il prof usa squeeze() che elimina le dimensioni ad 1
#(dal momento che sto passando una singola immagine).
```

```
def calc_hist_cv(img):
    return cv.calcHist([img], [0], None, [256], [0, 256]).squeeze()
```



calc_hist_py: 117 ms ± 4.38 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
calc_hist_np: 3.36 ms ± 17.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
calc_hist_cv: 121 µs ± 1.08 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

4.10 Analisi dell'istogramma

A livello di contenuto dell'immagine, quando non è troppo complessa, è possibile ricavare anche legate agli oggetti. Ad esempio si può notare la separazione fra una torre ed il cielo, questo tipo di istogramma è di tipo bimodale.

4.11 Operazioni sui pixel

Su una singola immagine

Formula $I'[y, x] = f(I[y, x])$

Consistono nell'applicazione di una funzione al valore di ciascun pixel. Ciò mi produrrà una immagine delle stesse dimensioni e il valore di ciascun pixel della nuova immagine dipenderà **unicamente** dal valore del corrispondente pixel nell'immagine di partenza. Esempi:

- Variazione della luminosità;
- variazione del contrasto;
- conversione da livelli di grigio a (pseudo) colori;
- binarizzazione con soglia globale.

Su più immagini

Formula $I'[y, x] = f(I_1[y, x], I_2[y, x], \dots)$

In questa applicazione vengono prese più immagini (tutte con le stesse identiche dimensioni), la nostra funzione prende il valore di un pixel e lo distribuisce fra più immagini, ritornando poi una singola immagine.

4.12 Variazione luminosità e contrasto

Formula classica

$$f(v) = \alpha \cdot v + \beta$$

v è un valore di luminosità per un'immagine, per capirci è quello che nella sezione precedente abbiamo chiamato $I[y, x]$. α controlla il contrasto, β controlla la luminosità e vogliamo che il valore di output sia un byte (per garantirlo si va a tagliare, andandolo a forzare nell'intervallo $[0, 255]$). Ponendo $\alpha = 1$ e $\beta = 0$ viene la funzione identità.

4.13 Gamma correction - funzione non lineare

Modo non lineare per aumentare la luminosità dei toni scuri o la luminosità dei toni chiari.

4.13.1 Formula

$$f(v) = \left(\frac{v}{255}\right)^{\gamma} \cdot 255$$

- se $\gamma < 1$ aumenta la luminosità dei toni scuri;
- se $\gamma > 1$ diminuisce la luminosità dei toni chiari;
- se $\gamma = 1$ otteniamo la funzione identità.

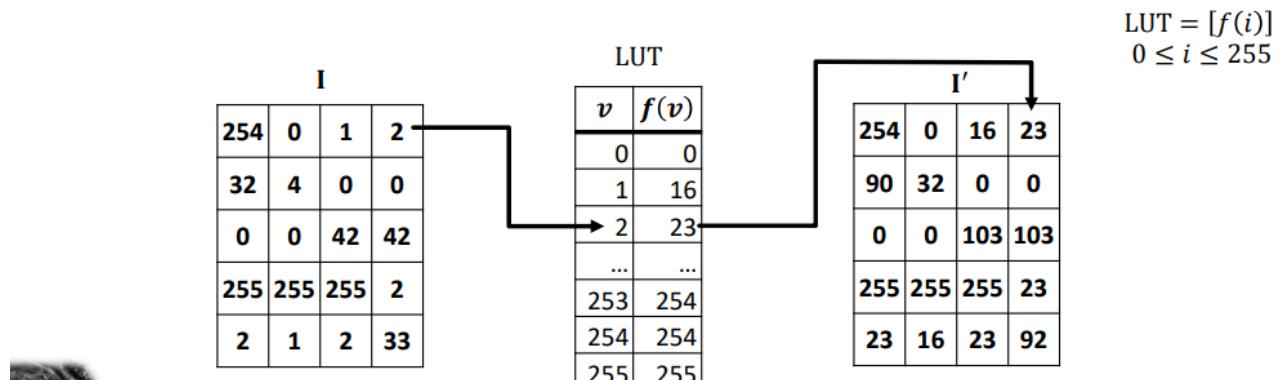
In prima istanza normalizziamo il numero di pixel fra 0 ed 1 (attraverso la frazione, 0 corrisponde al nero, 1 corrisponde al bianco). Anche elevando alla γ il valore resterà comunque compreso fra 0 e 1, moltiplicando per 255 infine otterrò un valore compreso nell'intervallo [0, 255].

4.14 Lookup table (LUT)

Concetto molto importante quando si lavora con funzioni che si applicano sul singolo pixel. In generale la lookup table rappresenta un concetto molto utilizzato in informatica, è una tabella in cui vado a cercare qualcosa che mi interessa. Idea di base: faccio il calcolo una volta e lo salvo nella table, in modo da evitare inutili sprechi computazionali. In questo caso ho in memoria un array di 256 byte, determinando applicando la funzione $f(v)$ su 256 valori distinti.

$$\mathbf{I}'[y, x] = f(\mathbf{I}[y, x]) \iff \mathbf{I}'[y, x] = \text{LUT}[\mathbf{I}[y, x]]$$

$$f(v) = \left(\frac{v}{255}\right)^{0.5} \cdot 255$$



4.15 LUT in python/numpy/OpenCV

```
# Un esempio di funzione f (Gamma correction per un certo valore di )
y = 0.5
# Calcolo di un singolo valore di f
f = lambda p: 255 * (p/255.0)**y
# Calcolo di f su tutti i valori di un array NumPy, impongo .astype(np.uint8)
# perche' altrimenti numpy restituisce un risultato in virgola
f_np = lambda a: f(a).astype(np.uint8)
# Calcolo dell'array LUT
lut = f_np(np.arange(256))
# Una semplice implementazione Python senza lookup table
def applica_py_f(img):
    res = np.empty_like(img) #creo una immagine vuota
    h, w = res.shape
    #applico la funzione f al singolo valore del pixel, ripetuto per tutti i pixel
    for y in range(h):      #vado da 0 fino ad h - 1
        for x in range(w):   #vado da 0 fino a w - 1
            res[y,x] = f(img[y,x])
    return res

# Semplice implementazione Python con LUT
def applica_py_lut(img):
    res = np.empty_like(img)
    h, w = res.shape
    for y in range(h):
        for x in range(w):
            res[y,x] = lut[img[y,x]]
    return res

# Implementazione NumPy senza LUT, applica la funzione f su tutto l'array, prima
# l'ho usato per calcolare la LUT ma nulla mi vieta di applicare questa
# funzione su un array bidimensionale (in questo caso la mia immagine).
def applica_np_f(img):
    return f_np(img)

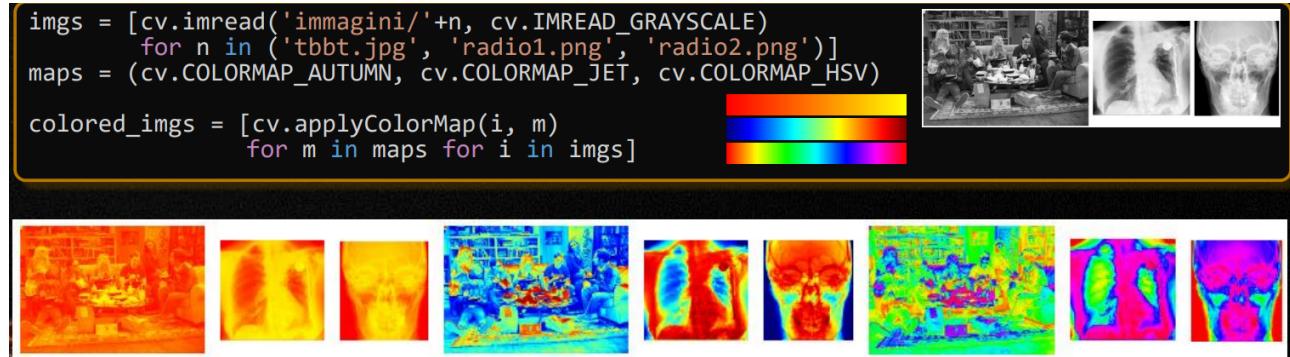
# Implementazione NumPy con LUT, sto sfruttando l'indicizzazione con img, un
# array bidimensionale di interi
def applica_np_lut(img):
    return lut[img]

# Funzione LUT di OpenCV
def applica_cv_lut(img):
    return cv.LUT(img, lut)
```

Metodo	Tempo (ms)	Speed-up
py_f	823,00	1
py_LUT	88,50	9
np_f	3,04	271
np_LUT	0,58	1419
cv_LUT	0,10	8230

4.16 LUT da grayscale ad RGB

Viene preso in input un array lungo 255 e in uscita ritorna un colore. La percezione umana non è adatta ad osservare piccole variazioni fra toni di grigio. I nostri occhi sono più sensibili a variazioni fra colori, inoltre in molte applicazioni si ricolorano immagini grayscale per renderle meglio fruibili. OpenCV mette a disposizione una funzione apposita (`applyColorMap`) e anche una serie di mappe di colori già pronte all'uso. La `colorMap` rimappa i colori, ricreando una scala che va da 0 a 255.



4.17 Operazioni aritmetiche fra immagini

4.17.1 Differenza

Una operazione comune è la sottrazione dello sfondo ed è molto utile in visione artificiale, permette di catturare ed analizzare porzioni di immagini. Nota: `np.nonzero` è una funzione che mi ritorna gli indici degli elementi che sono diversi da 0.

```
#Ho immagine e sfondo, separate.
img, back = cv.imread('esempi/mario-game.png'), \
             cv.imread('esempi/mario-back.png')
#Qui fa la differenza bit a bit
diff = img - back # N.B. diff = cv.subtract(img, back) e' piu' efficiente
#scelgo tutto cio' che non e' zero, di conseguenza rimane mario
mask = diff!=0
#creo una immagine tutta a zero e dopodiche' applico la maschera
res = np.zeros_like(img)
res[mask] = img[mask]
```



4.17.2 Operazioni bitwise

```
sprite = cv.imread('esempi/sprite.png')
mask = cv.imread('esempi/mask.png')
back = cv.imread('esempi/hill.jpg')
res = back.copy()
x, y = 200, 100
h, w = sprite.shape[:2]
#roi = region of interest, pezzettino dello sfondo
roi = res[y:y+h,x:x+w]
#questo e' l'AND bit a bit , se faccio l'and con bit a 1 non succede niente,
    rimane il valore che c'era prima mentre se faccio l'AND con valori a 0 si
    cancella tutto. Quindi facendo l'and con la maschera nello sfondo, sto
    andando a lasciare dove c'e' il bianco (riferito alla sprite) lo sfondo,
    altrimenti c'e' il nero
roi &= mask
#Facendo l'or bit a bit i valori a 0 rimangono tali, gli altri vengono cambiati
roi |= sprite
```

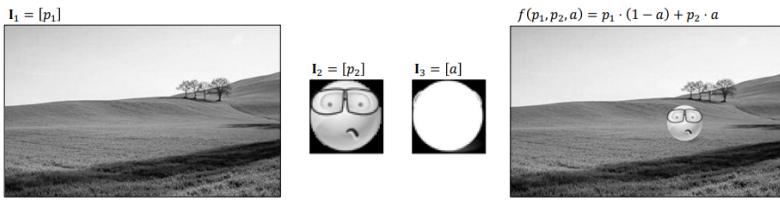


4.17.3 Alpha blending o aliasing

Per ottenere un risultato migliore rispetto alle operazioni bitwise ci serve un ulteriore canale, chiamato α . Si tratta di un canale che contiene un valore di trasparenza (di solito è un valore floating point fra 0 e 1) e la formula che vado ad applicare corrisponde ad una operazione fra tre immagini, la prima immagine è lo sfondo, la seconda immagine è la sprite e la terza immagine è il canale di trasparenza (è come una maschera, però invece che avere solo il valore 255 e 0, questa può assumere tutti i possibili colori). Come funziona? non si lavora più coi bit, si sfrutta una formula matematica che utilizza i numeri floating point.

```
img1 = cv.imread('esempi/hill.jpg')
img2 = cv.imread('esempi/study.png')
img2_alpha = cv.imread('esempi/study-alpha.png')
h, w = img2.shape[:2]
a = img2_alpha / 255 # Trasforma il canale alpha da [0,255] a [0,1]

x, y = 200, 100
res = img1.copy()
res[y:y+h,x:x+w] = img1[y:y+h,x:x+w]*(1.0-a) + img2*a
```



4.18 Binarizzare un'immagine grayscale in opencv

La binarizzazione consiste nel trasformare una immagine da grayscale ad una con gamma di due colori (generalmente bianco e nero).

4.18.1 Soglia globale

Il metodo più semplice per binarizzare consiste nell'utilizzo di una soglia globale. Una soglia globale è definita dalla seguente funzione:

$$f(v, t) = \begin{cases} 0 & v < t \\ 255 & v \geq t \end{cases}$$

Il problema dunque è concentrato nella identificazione della soglia... generalmente viene scelta osservando l'istogramma e tracciando una linea di demarcazione fra i due picchi di colore. La soglia si può individuare anche manualmente o attraverso dei metodi, come ad esempio il metodo di Otsu.

Il metodo di Otsu

Semplice algoritmo che cerca di trovare quella soglia che va a minimizzare la varianza intra-classe dell'intensità dei pixel delle due classi (una volta effettuata la binarizzazione distinguiamo pixel di background e foreground) determinate dalla soglia stessa.

4.18.2 Soglia locale

Non sempre una soglia globale può fare al caso nostro, ci sono immagini dove la luce varia in un modo per cui lo sfondo è molto più scuro che in altre. Per risolvere questo problema scelgo più soglie, ad esempio potrei dividere l'immagine in più parti e ad ognuna applicare una diversa soglia. Il metodo più generale per la soglia locale è quello di avere una soglia diverso per ciascun pixel.

Succo del discorso

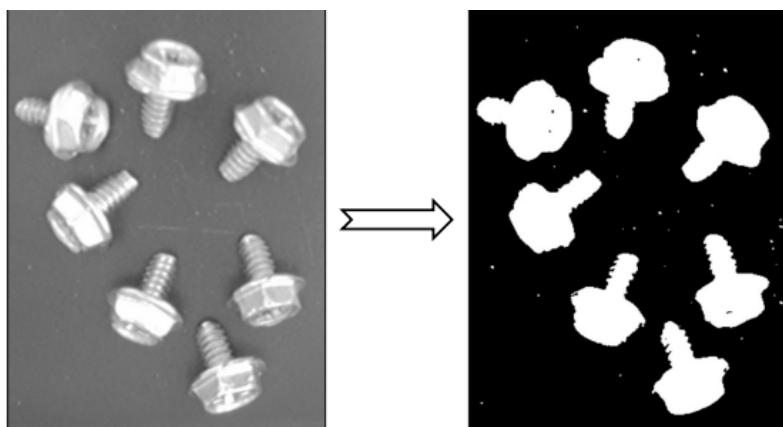
La soglia locale è determinata per ogni pixel considerando una piccola regione dell'immagine attorno ad esso. Il metodo più semplice per definirla consiste nel determinare la soglia come media dei pixel nella regione meno un valore costante (openCV ha anche un metodo che sfrutta l'intorno Gaussiano invece che la media).

4.18.3 Binarizzazione su OpenCV

OpenCV ci mette a disposizione la funzione threshold, che ci permette di binarizzare una immagine. I parametri che prende sono: l'immagine, il valore della soglia, il valore massimo ed una costante (corrisponde alla soglia globale, posso passare anche OTSU, che ignorerà i parametri). La funzione restituisce una tupla, formata da : valore di ritorno e immagine, il valore di ritorno è la soglia (può tornare utile quando uso OTSU). Per quanto riguarda la binarizzazione con soglia locale OpenCV mette a disposizione la funzione adaptiveThreshold, a cui passo (immagine; soglia; costante = tipo di soglia locale; costante = come utilizzo la soglia locale; dimensione della regione di pixel; costante = ciò che sottraggo alla media per ottenere la soglia).

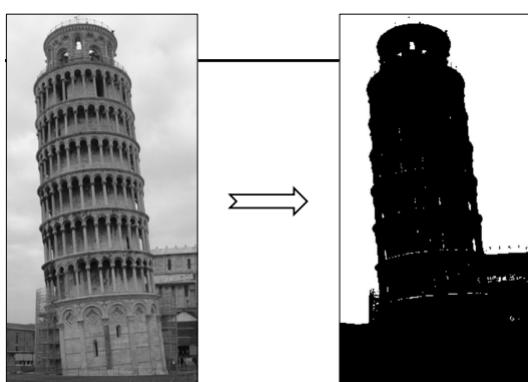
Chiudi

```
# Soglia globale (128)
img = cv.imread('esempi/bolts.png', cv.IMREAD_GRAYSCALE)
_, res = cv.threshold(img, 128, 255, cv.THRESH_BINARY)
```



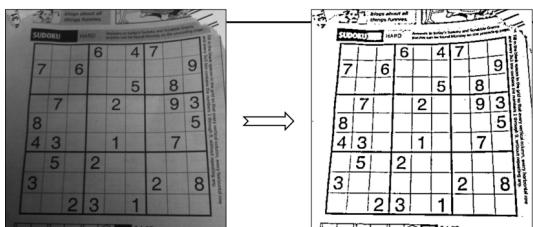
Torre

```
# Soglia globale determinata dall'algoritmo di Otsu
img = cv.imread('immagini/torre.jpg', cv.IMREAD_GRAYSCALE)
t, res = cv.threshold(img, -1, 255, cv.THRESH_OTSU)
```



Sudoku

```
# Soglia locale (media su intorno 11x11 meno il valore 10)
img = cv.imread('immagini/sudoku.jpg', cv.IMREAD_GRAYSCALE)
res = cv.adaptiveThreshold(img, 255, cv.ADAPTIVE_THRESH_MEAN_C,
cv.THRESH_BINARY, 11, 10)
```



Ferrari

Fino ad ora abbiamo binarizzato delle immagini grayscale, quando abbiamo immagini a colori? posso eventualmente binarizzare i singoli canali, sempre che l'operazione abbia senso. In hls potrebbe essere interessante binarizzare la saturazione, come nell'esempio sottostante.

```
# Cosa succede binarizzando il canale saturazione di un'immagine HSL?
img = cv.imread('esempi/ferrari.jpg')
hls = cv.cvtColor(img, cv.COLOR_BGR2HLS)
#qui ho sovrascritto la saturazione con la binarizzazione della stessa
_,hls[...,2] = cv.threshold(hls[...,2], 150, 255, cv.THRESH_BINARY)
#ho riconvertito da hls a bgr
res = cv.cvtColor(hls, cv.COLOR_HLS2BGR)
```



4.19 Contrast stretching

L'idea del contrast stretching è di migliorare una immagine poco contrastata, ampliando il contrasto, lo si fa espandendo i livelli di grigio.

Funzione

$$f(I[y, x]) = 255 \cdot \frac{I[y, x] - \alpha}{\beta - \alpha}$$

con α = "minimo livello di grigio dell'immagine" e β = "massimo livello di grigio dell'immagine". Togliendo il minimo e dividendo per il range cosa verrà fuori, sapendo che α è il minimo di tutti i pixel? Abbiamo che il numeratore della frazione, diviso per il range massimo mi darà sempre un numero compreso fra 0 e 1. Così facendo ho riscalato tutti i miei valori, andando ad allargarli.

Problema

E' una bella idea, peccato che normalmente non funzioni assolutamente, ma perchè? se c'è anche un solo pixel a zero ed uno solo a 255, la funzione diventa identità, non fa niente dunque. Come risolvo? Considero i bit molto scuri e molto chiari degli outlier, dunque non interessanti. Quindi questo mi permette di ridefinire α e β , che non necessariamente conterranno il valore minimo o massimo. Posso scegliere ad esempio di scartare il 5 per cento dei pixel più chiari ed il 5 per cento dei pixel più scuri.

4.20 Contrast stretching in Python/OpenCV

La formula è normalmente implementabile in numpy, l'unico accorgimento è fare in modo che tutti i valori siano float, per fare in modo di calcolare in floating point. Con np.clip, effettuo un troncamento e forzo il risultato chiudendo il range da 0 a 255. Per la trovata del percentile sfrutto la funzione np.percentile per definire α e β .

```
def contrast_stretching(img, a, b):

    # Converte in floating point
    n = 255*(img.astype(float)-a)/(b-a)

    # Forza il range [0,255] e converte in byte
    return np.clip(n, 0, 255).astype(np.uint8)

img = cv.imread('immagini/rice.png', cv.IMREAD_GRAYSCALE)

# Contrast stretching con alfa=min(I) e beta=max(I)
res = contrast_stretching(img, img.min(), img.max())
img = cv.imread('esempi/kernel.png', cv.IMREAD_GRAYSCALE)

# Contrast stretching con alfa=P_5(I) e beta=P_95(I)
res = contrast_stretching(img, np.percentile(img, 5), np.percentile(img, 95))
```

4.21 Equalizzazione in OpenCV

Altro modo per migliorare il contrasto, il vantaggio rispetto al contrast stretching è che non ha parametri, quindi gli passo l'immagine e fa quello che deve fare (non è detto che sia meglio rispetto allo stretching, dipende dal contesto). Si sfrutta spesso per migliorare il contrasto o rendere confrontabili immagini catturate in condizioni diverse di illuminazione.

4.21.1 Funzionamento

L'equalizzatore vuole distribuire tutti i livelli di grigio dell'istogramma in maniera uniforme. Zone dell'istogramma che sono scarsamente popolate, ovvero con pochi pixel, vengono compresse mentre le zone con molti pixel vengono allargate. Vogliamo dare più sfumature di grigio, aumentando il contrasto dove c'è più concentrazione di pixel e diminuendolo nelle zone meno popolate.

4.21.2 Formula

$$f(v) = \sum_{i=0}^v H[i]$$

Con H istogramma dell'immagine normalizzato: $H[i] = \frac{255 \cdot h[i]}{\sum h[i]}$ → $\sum_{i=0}^{255} H[i] = 255$

```
#Equalizzazione di un'immagine grayscale
img = cv.imread('esempi/kernel.png', cv.IMREAD_GRAYSCALE)
res = cv.equalizeHist(img)

# Equalizzazione di un'immagine a colori
# Esempio di come NON fare: equalizzazione di ciascun canale RGB
bgr = cv.imread('immagini/tbbt.jpg')
b, g, r = cv.split(bgr)
b_eq, g_eq, r_eq = [cv.equalizeHist(x) for x in (b, g, r)]
res_wrong = cv.merge((b_eq, g_eq, r_eq))

# Esempio di come procedere convertendo in HSL ed equalizzando L
hls = cv.cvtColor(bgr, cv.COLOR_BGR2HLS)
h, l, s = cv.split(hls)
l_eq = cv.equalizeHist(l)
hls_eq = cv.merge((h, l_eq, s))
res_ok = cv.cvtColor(hls_eq, cv.COLOR_HLS2BGR)
```

Capitolo 5

Calibrazione

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
    public void paintComponent(Graphics g) {
        g.drawString("Hello, world!", 65, 95);
    }
}
```

Capitolo 6

Filtri

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
    public void paintComponent(Graphics g) {
        g.drawString("Hello, world!", 65, 95);
    }
}
```

Capitolo 7

Analisi

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
    public void paintComponent(Graphics g) {
        g.drawString("Hello, world!", 65, 95);
    }
}
```

Capitolo 8

Movimento

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
    public void paintComponent(Graphics g) {
        g.drawString("Hello, world!", 65, 95);
    }
}
```

Capitolo 9

Riconoscimento

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
    public void paintComponent(Graphics g) {
        g.drawString("Hello, world!", 65, 95);
    }
}
```
