



**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

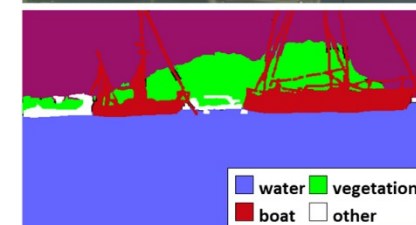
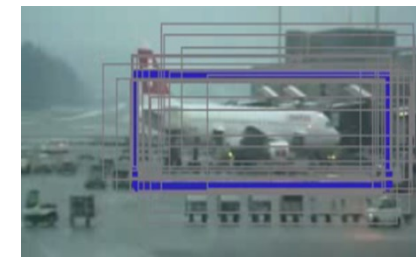
*Corso di Visione e Percezione  
A.A. 2019/2020*

# Liste in Python

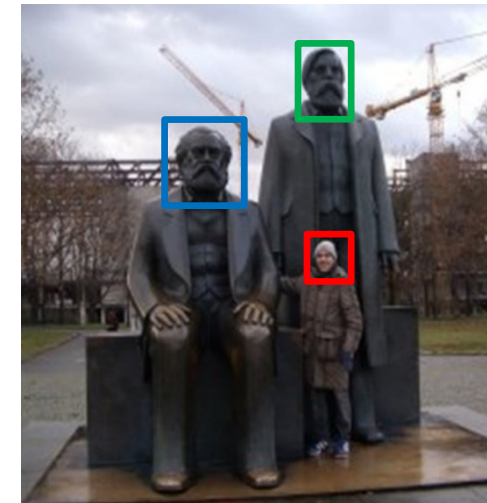
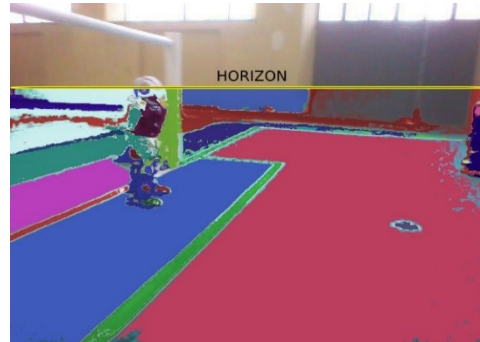
Docente  
**Domenico Daniele Bloisi**



Parte del materiale deriva dai corsi dei proff. Paolo Caressa e Raffaele Nicolussi (Sapienza Università di Roma) e Giorgio Fumera (Università degli Studi di Cagliari)



water vegetation  
boat other



Marzo 2020

# Il corso

---

- Home page del corso  
<http://web.unibas.it/bloisi/corsi/visione-e-percezione.html>
- Docente: Domenico Daniele Bloisi
- Periodo: **Il semestre** marzo 2020 – giugno 2020  
Martedì 17:00-19:00 (Aula GUGLIELMINI)  
Mercoledì 8:30-10:30 (Aula GUGLIELMINI)

# Obiettivi del corso

---

Il corso intende fornire agli studenti conoscenze relative alla **programmazione in Python** per lo sviluppo di applicazioni basate sul sistema operativo ROS, sulla libreria per la percezione OpenCV e sulla libreria per il Deep Learning Keras



<https://www.youtube.com/watch?v=l9KYJlLnEbw>

# Tipi semplici e strutturati

---

I tipi di dato possono essere classificati in:

1. tipi **semplici**
2. tipi **strutturati**

Un tipo semplice è composto da valori che non possono essere “scomposti” in valori più semplici ai quali sia possibile accedere attraverso operatori o funzioni del linguaggio. Esempi di tipi semplici del linguaggio Python (e di altri linguaggi) sono i numeri interi, i numeri frazionari e i valori Booleani.

Un tipo strutturato è invece composto da valori che sono a loro volta collezioni o sequenze di valori più semplici.

Le stringhe sono un esempio di tipo strutturato: sono infatti composte da sequenze ordinate di caratteri a cui è possibile accedere individualmente.

# Tipi di dato strutturati di Python

---

Oltre alle stringhe, due dei principali tipi strutturati del linguaggio Python sono:

1. le **liste**, che consentono di rappresentare sequenze ordinate di valori qualsiasi
2. i **dizionari**, che consentono di rappresentare collezioni (non ordinate) di valori qualsiasi

Esistono anche altri tipi di dato strutturati (come le **tuple**) ed è possibile definirne di nuovi.

# Il tipo di dato `list`

---

Le coordinate di un punto in un dato sistema di riferimento possono essere rappresentati come una sequenza ordinata di numeri reali.

Per rappresentare in modo compatto le coordinate di un punto in uno spazio a tre dimensioni può essere utilizzata una singola variabile di tipo `list` invece che tre variabili distinte di tipo `float`.

Il tipo `list` fornisce questa possibilità per sequenze ordinate di valori che possono appartenere a tipi qualsiasi, anche diversi tra loro.

# Rappresentazione `list`

---

Una lista si rappresenta nei programmi Python come:

- ✓ una sequenza ordinata di valori
- ✓ scritti tra parentesi quadre
- ✓ separati da virgole

Esempi:

`[7, -2, 4]`

→ lista composta da tre numeri interi

`[-5.3, 6, True]`

→ lista composta da un numero reale, un numero intero e un valore logico

`[]`

→ lista vuota

# Il tipo di dato `list`

---

Le liste, come i valori di qualsiasi tipo di dato Python, sono espressioni e possono quindi essere scritte in qualsiasi punto di un programma nel quale possa comparire una espressione.

È possibile:

- stampare una lista con l'istruzione `print`, per esempio:


```
print([7, -2, 4])
```

- assegnare una lista a una variabile, per esempio:

```
v = [7, -2, 4]
```



# Esempi list


 **list.ipynb** ☆

File Edit View Insert Runtime Tools Help


+ CODE + TEXT | ↑ CELL ↓ CELL

>

```
[1] print([7, -2, 4])
```

 [7, -2, 4]

```
[2] v = [7, -2, 4]
    print(v)
```

 [7, -2, 4]

# Valori degli elementi di una `list`

---

I valori degli elementi di una lista possono a loro volta essere indicati attraverso espressioni.

Per esempio, dopo l'esecuzione della seguente sequenza di istruzioni:

```
x = 2
y = -5
z = [x, y ** 2 + 1, x == 3]
```

la variabile `z` sarà associata alla lista `[2, 26, False]`

# Valori degli elementi di una `list`

---



```
x = 2  
y = -5  
z = [x, y ** 2 + 1, x == 3]  
print(z)
```

```
☐ [2, 26, False]
```

# list annidate

---

Poiché gli elementi di una lista possono essere valori di un tipo qualsiasi, possono a loro volta essere liste.

Si parla in questo caso di liste annidate, esattamente come nel caso di istruzioni composte (condizionale e iterativa) che contengano al loro interno altre istruzioni composte.

Per esempio, la seguente espressione produce come risultato una lista di quattro elementi: un numero intero, una stringa, una lista di due elementi, e un altro numero intero:

```
[-3, "abcd", ["a", 'b'], 10]
```

# Accesso agli elementi di una `list`

---

Il linguaggio Python mette a disposizione dei programmatori diversi operatori e funzioni predefinite per le liste.

In particolare, essendo le liste un tipo strutturato, alcuni operatori consentono l'accesso ai singoli elementi di una lista.

Il meccanismo di accesso si basa sul fatto che ogni elemento è identificato univocamente dalla sua posizione all'interno di una lista (si ricordi che una lista è una sequenza ordinata di valori).

La posizione di ciascun elemento di una lista è rappresentata in linguaggio Python attraverso un numero intero, detto indice.

Per convenzione, l'indice del primo elemento di una lista è 0, l'indice del secondo elemento è 1 e così via.

# Principali operatori `list`

---

sintassi	descrizione
<code>lista<sub>1</sub> == lista<sub>2</sub></code>	confronto ("uguale a")
<code>lista<sub>1</sub> != lista<sub>2</sub></code>	confronto ("diverso da")
<code>espressione in lista</code>	verifica della presenza di un valore in una lista
<code>espressione not in lista</code>	verifica dell'assenza di un valore in una lista
<code>lista<sub>1</sub> + lista<sub>2</sub></code>	concatenazione
<code>lista[indice]</code>	indicizzazione: accesso a un elemento
<code>lista[indice<sub>1</sub>:indice<sub>2</sub>]</code>	<i>slicing</i> (sezionamento): accesso a una sottosequenza di elementi di una lista

---

# Operatori di confronto

---

Gli operatori `==` e `!=` consentono di scrivere espressioni condizionali (il cui valore sarà `True` o `False`) consistenti nel confronto tra due liste.

Sintassi:

```
lista_1 == lista_2  
lista_1 != lista_2
```

dove `lista_1` e `lista_2` indicano espressioni che abbiano come valore una lista.

Semantica:

Due liste sono considerate identiche se sono composte dallo stesso numero di elementi e se ogni elemento ha valore identico a quello dell'elemento che si trova nella stessa posizione nell'altra lista.

# Operatori di confronto

---



```
v = [1, 2.0, "3"]  
t = [1, 2, 3]  
print(v == t)  
print(v != t)  
z = [1, 2, '3']  
print(v == z)  
v_1 = [1, 2.1, "3"]  
print(v_1 == z)
```



```
False  
True  
True  
False
```



# Gli operatori `in` e `not in`

---

Gli operatori `in` e `not in` consentono di scrivere espressioni condizionali che hanno lo scopo di verificare se un certo valore sia presente o meno all'interno di una lista.

Sintassi:

`espressione in lista`

`espressione not in lista`

**dove** `espressione` indica una qualsiasi espressione Python

Semantica:

se il valore di `espressione` è presente tra gli elementi di `lista`, allora l'operatore `in` restituisce il valore `True`; in caso contrario, restituisce `False`.

Il comportamento dell'operatore `not in` è l'opposto di quello per `in`.

La ricerca non viene estesa agli elementi di eventuali liste annidate all'interno di `lista`.

# Gli operatori `in` e `not in`

---



```
v = [2, 5, 8]
print(4 in v)
print(4 not in v)
v_1 = [2, [5, 4], 8]
print(4 in v_1)
print(4 not in v_1)
print([5, 4] in v_1)
```



```
False
True
False
True
True
```

# L'operatore di concatenazione

---

L'operatore di concatenazione per le liste è analogo al corrispondente operatore del tipo di dato stringa

Sintassi:

```
lista_1 + lista_2
```

Semantica:

la concatenazione restituisce una nuova lista composta dagli elementi di `lista_1` seguiti da quelli di `lista_2`, disposti nello stesso ordine in cui si trovano nelle due liste.

Le liste originali non vengono modificate.

# L'operatore di concatenazione

---



```
v = [3, "otto", 12]
t = ["abc", 37]
r = v + t
print(r)
print(t + v)
```



```
[3, 'otto', 12, 'abc', 37]
['abc', 37, 3, 'otto', 12]
```

# L'operatore di indicizzazione

---

L'operatore di indicizzazione consente di accedere a ogni singolo elemento di una lista, per mezzo dell'indice corrispondente.

Sintassi:

```
lista[indice]
```

dove `indice` deve essere una espressione il cui valore sia un intero compreso tra 0 e la lunghezza della lista meno uno.

Semantica:

il risultato è il valore dell'elemento di lista il cui indice è pari al valore di `indice`. Se il valore di `indice` non corrisponde a una delle posizioni della lista si otterrà un messaggio di errore.

# Accesso agli elementi di una `list`



```
x = 2
y = -5
z = [x, y ** 2 + 1, x == 3]
print(z)
print(z[1])
print(z[11]).
```



```
[2, 26, False]
26
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-6-d785bfa749d2> in <module>()
      4 print(z)
      5 print(z[1])
----> 6 print(z[11])

IndexError: list index out of range
```

SEARCH STACK OVERFLOW

# Indici negativi per `list`

---



```
s = [3, 'A', 12.4, 3, 'B']  
print(s[-1])  
print(s[-2])
```



```
B  
3
```

L'indice -1 indica l'ultimo elemento della lista, -2 il penultimo e così via

# Modifica degli elementi di una `list`

---

L'operatore di indicizzazione consente anche di modificare i singoli elementi di una lista, attraverso una istruzione di assegnamento.

Sintassi:

```
lista[indice] = espressione
```

dove `lista` indica il nome di una variabile alla quale sia stata in precedenza assegnata una lista, mentre `espressione` indica una qualsiasi espressione Python

Semantica:

l'elemento di `lista` nella posizione corrispondente a `indice` viene sostituito dal valore di `espressione`.



# Modifica degli elementi di una `list`

---



```
v = [5, 6, 7]  
print(v)  
v[2] = 8  
print(v)
```



```
[5, 6, 7]  
[5, 6, 8]
```

# L'operatore di *slicing*

---

L'operatore di slicing restituisce una lista composta da una sottosequenza della lista a cui viene applicato.

Sintassi:

```
lista[indice_1:indice_2]
```

dove `indice_1` e `indice_2` sono espressioni i cui valori devono essere numeri interi compresi tra 0 e la lunghezza di lista.

Semantica:

il risultato è una lista composta dagli elementi di `lista` aventi indici da `indice_1` a `indice_2 - 1` (si noti che l'elemento avente indice pari a `indice_2` non viene incluso nel risultato).

Anche in questo caso la lista originale non viene modificata.

# L'operatore di *slicing*



```
v = [12, 34, 56, 78, 90, 13, 24, 35, 46, 67, 89]
s = v[2:5]
r = v[3:9]
print(v)
print(s)
print(r)
print(v[2:])
print(v[:2])
```



```
[12, 34, 56, 78, 90, 13, 24, 35, 46, 67, 89]
[56, 78, 90]
[78, 90, 13, 24, 35, 46]
[56, 78, 90, 13, 24, 35, 46, 67, 89]
[12, 34]
```



# Copia di una lista

---

Possiamo ottenere la copia di una lista usando il range [:]



```
r = 5
lista = [r, 't', 'z', 23, 56]
r = 7
print(lista)
copia = lista[:]
print(copia)
```

```
[5, 't', 'z', 23, 56]
[5, 't', 'z', 23, 56]
```

# Funzioni built-in per le `list`

---

funzione	descrizione
<code>len(lista)</code>	restituisce la lunghezza di una lista
<code>min(lista)</code>	restituisce l'elemento più piccolo in una lista composta da numeri
<code>max(lista)</code>	restituisce l'elemento più grande in una lista composta da numeri
<code>list(range(a))</code>	<code>a</code> deve essere un intero; se <code>a &gt; 0</code> restituisce la lista <code>[0, 1, ..., a-1]</code> , altrimenti restituisce una lista vuota
<code>list(range(a, b))</code>	<code>a</code> e <code>b</code> devono essere interi: se <code>a &lt; b</code> , restituisce la lista <code>[a, a+1, ..., b-1]</code> , altrimenti restituisce una lista vuota

---

# Funzioni built-in per le `list`

---



```
v = [12, 34, 56, 78, 90, 13, 24, 35, 46, 67, 89]
print(len(v))
print(max(v))
print(min(v))
print(list(range(12)))
print(list(range(3,9)))
```



```
11
90
12
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
[3, 4, 5, 6, 7, 8]
```

# Esempio: costruzione di una lista



```
lista = []
print ("Inserire cinque numeri.")
k = 1
while k <= 5:
    elemento = input("valore " + str(k) + ": ")
    lista = lista + [elemento]
    k = k + 1
print ("La lista inserita è:\n", lista)
```



```
Inserire cinque numeri.
valore 1: 4
valore 2: 5
valore 3: 6
valore 4: 7
valore 5: 0
La lista inserita è:
['4', '5', '6', '7', '0']
```

# Esempio: costruzione di una lista



```
lista = []  
print ("Inserire cinque numeri.")  
k = 1  
while k <= 5:  
    elemento = input("valore " + str(k) + ": ")  
    lista = lista + [elemento]  
    k = k + 1  
print ("La lista inserita è:\n", lista)
```



```
Inserire cinque numeri.  
valore 1: 4  
valore 2: 5  
valore 3: 6  
valore 4: 7  
valore 5: 0  
La lista inserita è:  
['4', '5', '6', '7', '0']
```



# Sequenze

---

Le **stringhe** e le **liste** sono tipi di dato che hanno in comune il fatto di essere costituite da sequenze ordinate di un numero qualsiasi di elementi.

Per questo motivo sono entrambe indicate in linguaggio Python con il termine più generale di **sequenze**.

Alcuni operatori e alcune funzioni built-in che operano su sequenze ordinate possono essere applicati sia alle liste che alle stringhe:

- funzioni built-in: `len`
- operatori: `in` e `not in`, indicizzazione, slicing, concatenazione

# Liste vs stringhe

---

Liste e stringhe presentano una importante differenza.

Mentre attraverso l'istruzione di assegnamento e l'operatore di indicizzazione è possibile modificare i singoli elementi di una lista, non è invece possibile modificare i singoli caratteri delle stringhe.

Per tale motivo,

- le **liste** sono dette **sequenze mutabili**
- Le **stringhe** sono dette **sequenze immutabili**  
Il tentativo di modificare un elemento di una stringa produce un errore!

# Liste vs stringhe

```
s = 'str'
print(len(s))
print('t' in s)
s2 = ['s', 't', 'r']
print(len(s))
print('t' in s2)
s2[1] = 'v'
print(s2)
s[1] = 'v'
```

```
3
True
3
True
['s', 'v', 'r']
```

-----

**TypeError** Traceback (most recent call last)  
[<ipython-input-27-3fdf0ee3315c>](#) in <module>()  
7 s2[1] = 'v'  
8 print(s2)  
----> 9 s[1] = 'v'

**TypeError:** 'str' object does not support item assignment

SEARCH STACK OVERFLOW

# Accesso agli elementi di liste annidate

---

Gli elementi di una lista possono essere valori di tipi qualsiasi, quindi anche strutturati, come stringhe o altre liste.

L'operatore di indicizzazione consente di accedere anche agli elementi di strutture annidate.

Se  $s$  è una variabile a cui è stata assegnata una lista e l'elemento di indice  $i$  della lista è a sua volta una sequenza (lista o stringa), sarà possibile accedere all'elemento di indice  $j$  di quest'ultima con la seguente sintassi:

$s[i][j]$

# Accesso agli elementi di liste annidate

---

```
[34] v = [3, [4, 5], 6, 10]
      print(v[1][1])
      print(4 in v)
      print(4 in v[1])
      v[1] = v[1]+[5]
      print(v)
      v[1][2] = 7
      print(v)
```



```
5
False
True
[3, [4, 5, 5], 6, 10]
[3, [4, 5, 7], 6, 10]
```

# Matrici

---

Una matrice di m righe e n colonne può essere rappresentata in un programma Python in diversi modi.

Una possibilità consiste nel memorizzare i suoi elementi, in un ordine opportuno, in una lista “semplice” (non annidata) di  $m \times n$  elementi.

Si consideri per esempio la seguente matrice:

$$\begin{pmatrix} -3 & 1 & 4 \\ 2 & 5 & -1 \end{pmatrix}$$

essa può essere rappresentata dalla lista:

`[-3, 1, 4, 2, 5, -1]`

# Matrici come liste annidate

---

La matrice considerata in precedenza

$$\begin{pmatrix} -3 & 1 & 4 \\ 2 & 5 & -1 \end{pmatrix}$$

può essere rappresentata dalla seguente lista nidificata:

`[[-3, 1, 4], [2, 5, -1]]`

Assumendo che tale lista sia stata assegnata a una variabile di nome `M`, per accedere all'elemento nella seconda riga ( $i = 2$ ) e nella prima colonna ( $j = 1$ ) si userà l'espressione `M[1][0]`

# Stampa di una matrice

---

```
def stampa_matrice(mat):  
    r = 0  
    while r < len(mat):  
        c = 0  
        while c < len(mat[r]):  
            print(str(mat[r][c]), end=' ')  
            c = c + 1  
        print('\n')  
        r = r + 1
```



# Stampa di una matrice



```
M = [[-3, 1, 4], [2, 5, -1]]
```

```
def stampa_matrice(mat):  
    r = 0  
    while r < len(mat):  
        c = 0  
        while c < len(mat[r]):  
            print(str(mat[r][c]), end=' ')  
            c = c + 1  
        print('\n')  
        r = r + 1  
  
stampa_matrice(M)
```



```
-3 1 4
```

```
2 5 -1
```

# L'istruzione iterativa `for`

---

Python include una versione alternativa dell'istruzione iterativa `while`: l'istruzione `for`

L'istruzione `for` consente di esprimere un solo tipo di iterazione che consiste nell'accedere a tutti gli elementi di una sequenza (stringa o lista)

L'accesso avviene dal primo all'ultimo elemento e non è possibile modificare tale ordine

# for: sintassi

---

```
for v in s
    sequenza_di_istruzioni
```

- `v` è essere il nome di una variabile
- `s` è una espressione avente come valore una sequenza (una lista o una stringa)
- `sequenza_di_istruzioni` è una sequenza di una o più istruzioni qualsiasi, che devono rispettare la regola sui rientri già vista per l'istruzione `while`

# for: esempio

---



```
sequenza = input("Inserire una stringa: ")  
print ("I suoi caratteri sono:")  
for elemento in sequenza:  
    print(elemento)
```

```
↳ Inserire una stringa: sequenza  
I suoi caratteri sono:  
s  
e  
q  
u  
e  
n  
z  
a
```

# for: stampa di una matrice

---



```
M = [[-3, 1, 4], [2, 5, -1]]
```

```
def stampa_matrice_for(mat):  
    for r in mat:  
        for c in r:  
            print(str(c), end=' ')  
        print('\n')
```

```
stampa_matrice_for(M)
```

```
[-3 1 4
```

```
2 5 -1
```

# Confronto tra `while` e `for`



```
M = [[-3, 1, 4], [2, 5, -1]]
```

```
def stampa_matrice(mat):  
    r = 0  
    while r < len(mat):  
        c = 0  
        while c < len(mat[r]):  
            print(str(mat[r][c]), end=' ')  
            c = c + 1  
        print('\n')  
        r = r + 1  
  
stampa_matrice(M)
```

```
[-3 1 4
```

```
2 5 -1
```



```
M = [[-3, 1, 4], [2, 5, -1]]
```

```
def stampa_matrice_for(mat):  
    for r in mat:  
        for c in r:  
            print(str(c), end=' ')  
        print('\n')  
  
stampa_matrice_for(M)
```

```
[-3 1 4
```

```
2 5 -1
```

# Confronto tra `while` e `for`

---

Non è possibile usare l'istruzione `for` con lo schema visto in precedenza per eseguire operazioni sugli elementi di una sequenza che richiedano l'uso esplicito degli indici, come per esempio:

- la modifica di un elemento di una lista attraverso una istruzione di assegnamento `lista[k] = valore`
- l'accesso a più di un elemento di una lista o di una stringa, per esempio per confrontare i valori di due elementi adiacenti con una espressione condizionale come la seguente:

```
lista[k] != lista[k + 1]
```

# Confronto tra `while` e `for`

---

Il ciclo `while` seguente assegna il valore 0 a tutti gli elementi di una lista precedentemente memorizzata in una variabile di nome `lista`:

```
k = 0  
  
while k < len(lista):  
    lista[k] = 0  
    k = k + 1
```

Se invece si usasse una sequenza di istruzioni come:

```
for elemento in lista:  
    elemento = 0
```

la lista **non** verrebbe modificata, poiché la variabile `elemento` contiene solo una **copia** del valore di ciascun elemento della lista



# Funzione enumerate

---

Grazie alla funzione `enumerate` è possibile usare l'istruzione `for` per accedere agli elementi di una sequenza usando una variabile come indice.



```
s = [1, 4, 2, 5]
print(s)
for count, elem in enumerate(s):
    s[count] += 3
print(s)
```



```
[1, 4, 2, 5]
[4, 7, 5, 8]
```

# metodo `split`

---

La funzione della classe `string` denominata `split` è molto utile per l'elaborazione di stringhe

Sintassi:

```
stringa.split()
```

dove `stringa` indica una variabile avente per valore una stringa.

La funzione `split` suddivide una stringa in corrispondenza dei caratteri di spaziatura (incluso il newline) e restituisce le corrispondenti sottostringhe all'interno di una lista, senza includere i caratteri di spaziatura.

La stringa originale non viene modificata.

# metodo split

---



```
stringa = "questa è una stringa"  
v = stringa.split()  
print(v)
```

```
☞ ['questa', 'è', 'una', 'stringa']
```

# metodo `split` per dati formattati

---

Se si desidera suddividere una stringa in corrispondenza di una sequenza di uno o più caratteri specifici, tale sequenza dovrà essere indicata (sotto forma di una stringa) come argomento di `split`

Sintassi:

`stringa.split(caratteri)`



```
s = "3,45,6,a,b,c,23"  
v1 = s.split()  
print(v1)  
v2 = s.split(',')  
print(v2)
```



```
['3,45,6,a,b,c,23']  
['3', '45', '6', 'a', 'b', 'c', '23']
```



**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

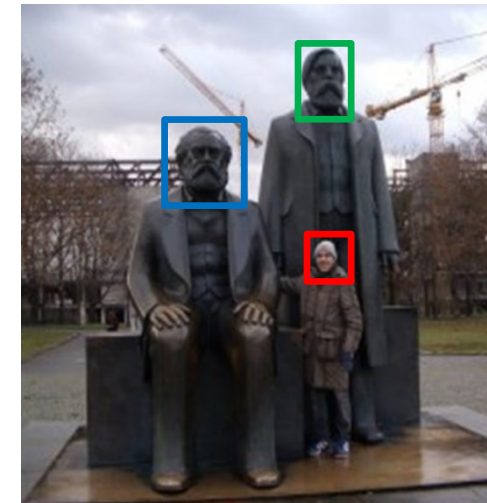
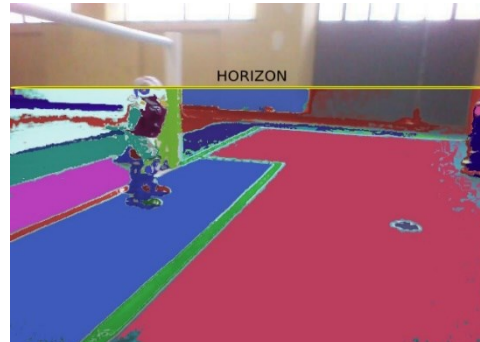
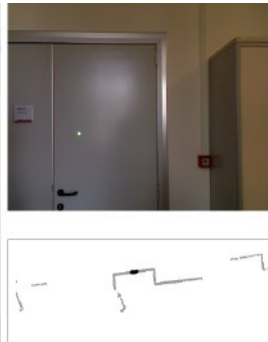
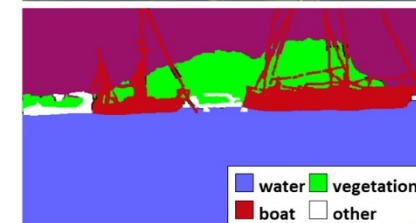
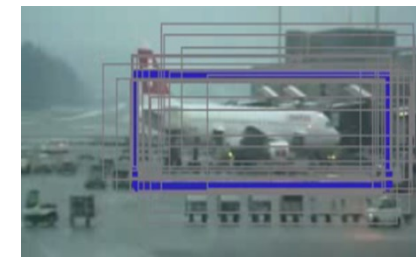
*Corso di Visione e Percezione  
A.A. 2019/2020*

# Liste in Python

Docente  
**Domenico Daniele Bloisi**



Parte del materiale deriva dai corsi dei proff. Paolo Caressa e Raffaele Nicolussi (Sapienza Università di Roma) e Giorgio Fumera (Università degli Studi di Cagliari)



Marzo 2020