

# **Fondamenti di Programmazione Robotica e Automatica**

**Introduzione alla programmazione**  
(tramite elaborazione di immagini)

Prof. Vincenzo Grassi

Rielaborate da Matteo Esposito

# Introduzione alla programmazione

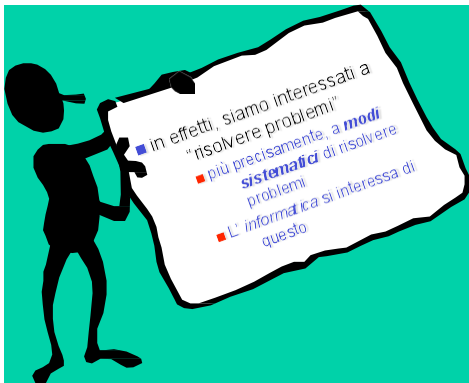
## ■ Obiettivi di apprendimento

### ■ generali

- utilizzazione dell'ambiente JES per lavorare con Python
- introduzione ai *tipi di dato*
- creare e usare *variabili*, per conservare valori e oggetti
- creare e usare *funzioni*
- scrivere algoritmi *sequenziali*, e codificarli in funzioni

### ■ casi di studio

- visualizzare immagini
- riprodurre suoni





python<sup>TM</sup>

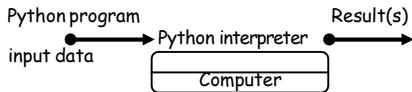


## linguaggi *interpretati* vs. linguaggi *compilati*

- Python è un linguaggio *interpretato*

- **interprete**: strato sw che simula il **processor** per un dato linguaggio

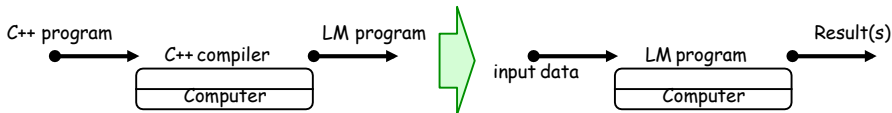
- esecuzione diretta (in un passo)



- altri linguaggi (p.es. Java, C++, ...) sono linguaggi *compilati*

- **compilatore**: **traduttore** da un linguaggio al linguaggio macchina (LM) di qualche computer "reale"

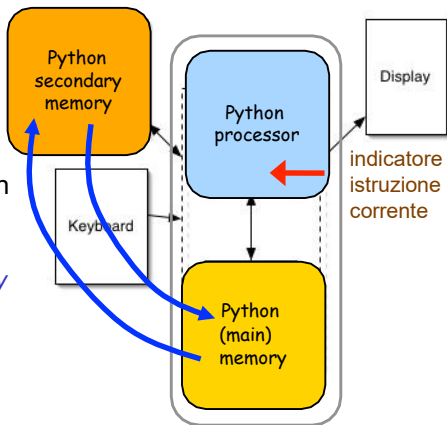
- processo di esecuzione in due passi



## “macchina Python”: architettura interna

- una vista “astratta” (a livello Python)
  - il *processor* esegue “operazioni” Python
  - le due *memorie* immagazzinano “entità”

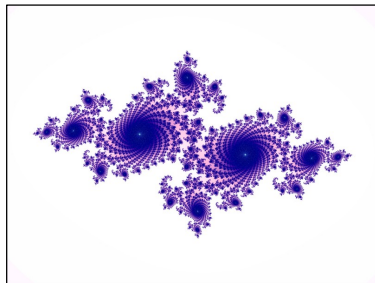
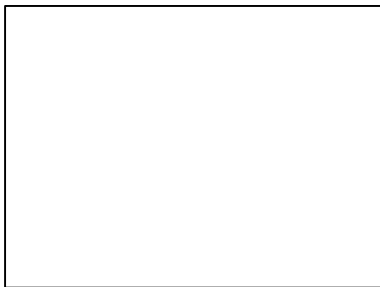
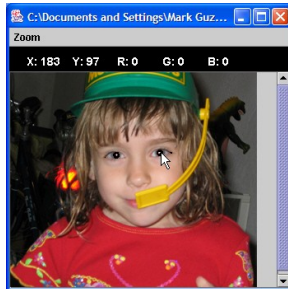
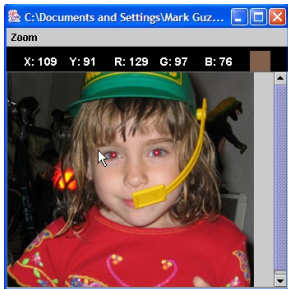
- il *processor* può lavorare SOLO con cose nella *main memory*
  - necessità di trasferire oggetti da *secondary memory* a *main memory* per lavorare con essi
  - ... e riportarli indietro se non volete perderli !











■ cosa ci serve sapere per risolvere **questi** problemi ?

■ come è **rappresentata** una immagine

■ quali **operazioni** sono disponibili

■ operazioni di base

+

■ modi per combinarle tra loro

**algoritmo**

---

■ come è strutturata la **macchina** che utilizzeremo

■ architettura interna

■ linguaggio

## Tipi di dato

- quali entità (valori) possono essere rappresentate e immagazzinate in memoria?

Python  
(main  
memory

- “valori semplici”

31

12

Interi (`int`)

-12

'Mark'

"Barbara Ericson"

Stringhe (`str`)

"85 5th Street NW"

34654.01

12.998

Reali (`float`)

1.01

0.01

- “valori complessi”

Immagini (`Picture`)



- ... e altro ancora ...

- programmi, ...

Nel computer, questi sono  
*tutti* nient'altro che bit

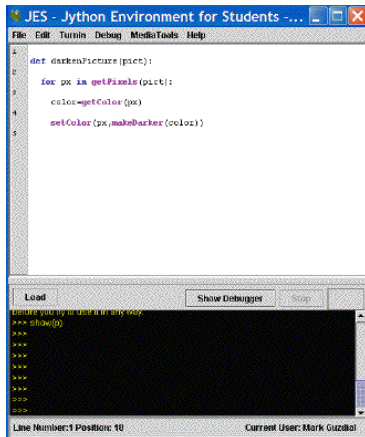
# iniziamo a usare JES : installazione

- **JES: Jython Environment for Students**
- **Installare e avviare JES**
  - andare su <https://github.com/gatech-csl/jes/releases/tag/5.020> e scaricare la versione di JES per il proprio computer
    - versione più recente: 5.02
    - per Linux, MacOS, Windows
    - assicurarsi di avere già installato un compilatore Java
      - MacOS: in caso di problemi, seguire le istruzioni nel file "JES Information.txt" scaricato insieme all'applicazione



## iniziamo a usare JES

- *Program area* : un semplice *editor* (per inserire i tuoi *programmi* o *algoritmi*)
- *Command area* : comandi che vogliamo far eseguire dalla macchina Python.



- suggerimento: usate il menu *Help*
  - grande quantità di validi consigli

# iniziamo a usare JES

- JES con *help* visualizzato
  - usate il menu “Window Layout” per ottenere la vista che preferite

The screenshot shows the JES application window titled "JES - Jython Environment for Students - incrRed". The menu bar includes File, Edit, Watcher, MediaTools, JES Functions, Window Layout, and Help. The main interface is divided into three sections:

- Program Area:** Contains a Python function definition:

```
1 def increaseRed(picture):  
2     for p in getPixels(picture):  
3         setRed(p, getRed(p) * 1.5)  
4
```
- Command Area:** A black area with white text showing a REPL session:

```
>>> print 34 + 56  
90  
>>> print 1/2  
0  
>>>
```
- Help Area:** A white area on the right showing the documentation for the `setRed` function:

```
setRed(pixel, redValue):  
pixel: the pixel you want to set the red  
value in.  
redValue: a number (0 - 255) for the new  
red value of the pixel  
Takes in a Pixel object and a value  
(between 0 and 255) and sets the redness  
of that pixel to the given value.  
Example:  
def zeroRed(pixel):  
    setRed(pixel, 0)  
This will take in a pixel and set its amount
```

At the bottom of the window, there is a status bar with the text `setRed(pixel, redValue)`, a button labeled "Explain setRed", and the text "Line Number:3 Position: 9". An arrow points from the text "Help Area" below the screenshot to the Help Area section.

Help Area

## Usare JES

- ogni volta che scrivete un *comando/espressione* nella *command area* e premete il tasto <return>, l'interprete Python lo *esegue/valuta*
  
- iniziare con JES ...
  - il *prompt* >>>
  - esempi di valutazione di espressioni

# Usare JES

- valutazione di semplici espressioni
  - provatele con l'interprete JES

>>> 34 + 56            **somma di interi**

>>> 34.1/46.5        **divisione di reali**

>>> 22 \* 33           **moltiplicazione di interi**

>>> 14 - 15           **sottrazione di interi**

>>> "Hello" + "Mark"   **somma (concatenazione) di due stringhe**

>>> sin(0)            **funzioni trigonometriche**

>>> cos(0)

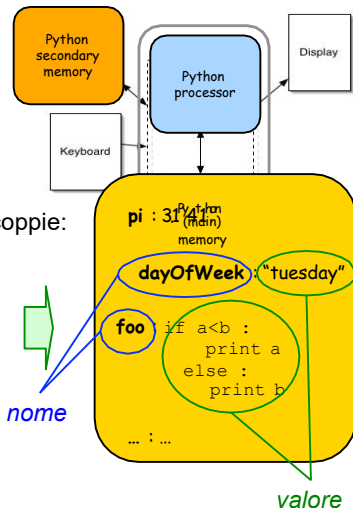
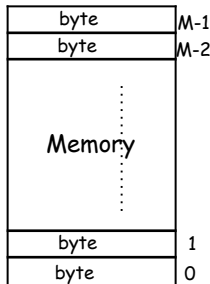
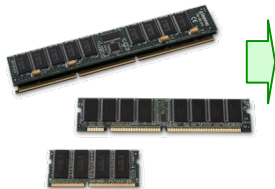
>>> log(2.7)           **funzioni matematiche**

>>> 2\*cos(0.5) + sqrt(4)



# “cose” in memoria

- quale genere di “cose” (dati) ?
  - abbiamo visto qualche esempio
- come fare riferimento ad esse?
  
- per usarle, dobbiamo dare loro un **nome**
  - la memoria (di Python) come un insieme di coppie: **[nome, valore]**
  - è una **astrazione**



**[*nome, valore*]**

***“statico”***



**variabile**

## “cose” in memoria

- Programmazione : *dare nome* alle cose (+ altro ...)

- *ai dati*

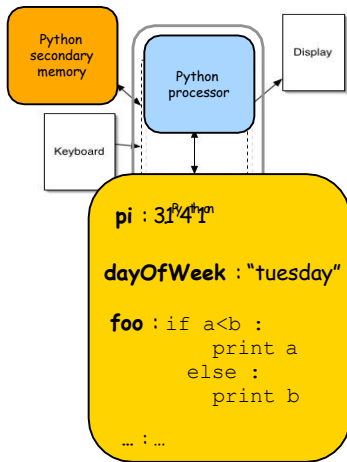
- ai “valori” che manipoliamo
  - [nome+valore] : **variabile**

- *agli algoritmi*

- codificati in Python
  - [nome+algoritmo] : **funzione**

- La “qualità” dei nomi è importante (come in filosofia o matematica)

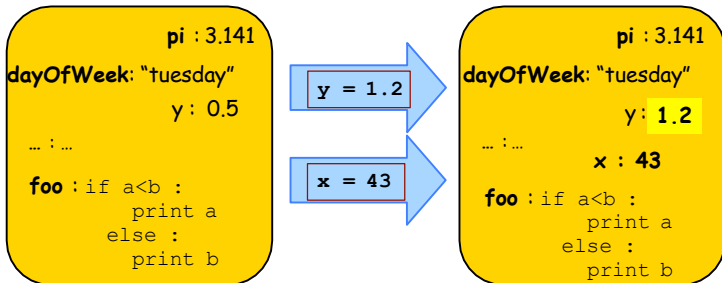
- descrittivi
- comprensibili



## costruzione di variabili (coppie [nome, valore] )

- il nostro primo *comando* Python
- l'operatore di *assegnamento* `=`
  - serve a costruire coppie [nome, valore] dove *valore* è un "dato"
  - non si usa quando *valore* è un algoritmo codificato in Python
    - operatore *differente*
- sintassi: `nome = espressione`

- *semantica*:



## nomi validi

- Possiamo usare (quasi) tutti i nomi che vogliamo, ma ...
- ... devono iniziare con una *lettera* o con il carattere `_`
  - seguiti da una qualsiasi combinazione di caratteri *alfanumerici*, più il carattere `_`
- *Maiuscole/minuscole sono distinte*
  - Print *non è la stessa cosa di* print
  - myPicture *non è la stessa cosa di* mypicture
- Attenzione a non usare nomi Python
  - Le *parole riservate* di Python
    - and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, lambda, not, or, pass, print, raise, return, try, while, yield

## questioni di stile

- convenzioni adottate nei nomi delle variabili
  - scelta libera (ovviamente ...), ma esistono alcune convenzioni standard che è bene rispettare
  - per esempio: *notazione a cammello (camel case)*

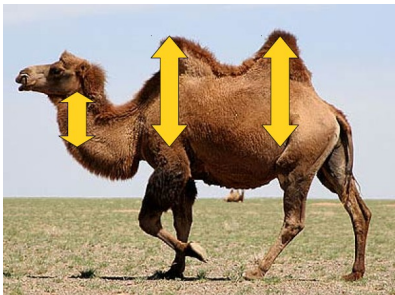
tutteLeInizialiMaiuscoleTranneLaPrima



## questioni di stile

- convenzioni adottate nei nomi delle variabili
  - scelta libera (ovviamente ...), ma esistono alcune convenzioni standard che è bene rispettare
  - per esempio: **notazione a cammello (camel case)**

tutteLeInizialiMaiuscoleTranneLaPrima



## questioni di stile

- convenzioni adottate nei nomi delle variabili
  - scelta libera (ovviamente ...), ma esistono alcune convenzioni standard che è bene rispettare
  - per esempio: *notazione a cammello (camel case)*

tutteLeInizialiMaiuscoleTranneLaPrima

- nomi descrittivi
  - se ci si riferisce al numero di votanti, scegliete: numeroDiVotanti, oppure numVotanti, oppure votanti
  - non usate x, oppure controlloVolume, oppure zioPaperone



## Usare JES

- esempi di creazione e uso di **variabili**
  - ... *provate*

>>> value = 12      **Definire un variabile con valore intero**

>>> value \* 3      **Usare una variabile in un'espressione**

>>> value      **Il valore della variabile non è cambiato!!!**

>>> name = "Mark"      **Definire un variabile con valore di tipo string**

>>> name \* 3      **Usare una variabile in un'espressione**

>>> value = 20      **Solo il comando "=" può cambiare il valore di una variabile**

>>> value \* 3

>>> value = "John"      **E' possibile cambiare il *tipo* di valore associato con un nome**

>>> x = 10

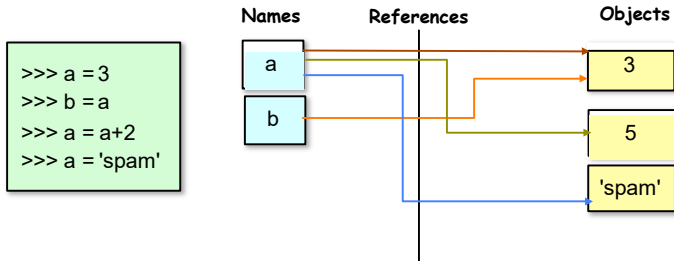
>>> x = x - 3      **Che cosa significa?**

## Usare JES: consigli pratici

- lavorare nella *command area*
  - le frecce *up/down* possono essere usate per scorrere lungo la *command history*
  - Si può modificare l'ultima riga di comando
    - posizionate il cursore alla fine della riga , prima di premere il tasto *Return/Enter*.
    - si può modificare la riga a piacere, anche utilizzando *Cut/Copy/Paste*

## ancora sul comando “=”

- valori “condivisi”
  - ogni volta che il comando “=” viene eseguito ...
    - si crea un *nome* (a meno che non esista già), e lo si fa riferire a un *oggetto* (che diventa il *valore* associato con quel *nome*)
  - nomi differenti possono essere associati con lo stesso valore
  - in questo caso, modificare uno di essi (il valore associato ad esso) non ha alcun effetto sugli altri



?

```
>>> a = 10
>>> b = a
```

Quale delle affermazioni sotto è vera dopo che sono stati eseguiti questi due comandi ? (potrebbe esserlo più di una)

- 1) La variabile `a` ora è indefinita
- 2) La variabile `a` vale ancora 10
- 3) La variabile `b` ora vale 10
- 4) Se cambiamo ancora il valore di `a`, cambierà anche quello di `b`

1

2

3

4

## Usare JES

- un altro comando Python : `print`

- **semantica**: valuta *expression* e visualizza il suo valore (se ce n'è uno)
- quale differenza con lo scrivere soltanto: *expression* ?
- ... provate

```
>>> x = 20
>>> x + 3
>>> print x + 3
```

```
>>> name = 'Mary'
>>> 2*name
>>> print 2*name
```

- `>>> expression` : Python valuta e visualizza la rappresentazione *interna* del valore di *expression*
- `>>> print expression` : Python valuta e visualizza la rappresentazione *stampabile* (esterna) del valore di *expression*
- spesso le due cose sembrano identiche, ma a volte non lo sono
  - p.es., per stringhe

- generalizzazione: `print`

## Python 2.x vs. Python 3.x

- `print` è un comando Python 2.x

- la *versione* di Python utilizzata in JES

- versione 2.6

- Python 3.x usa una sintassi diversa: `print( )`

- o, in generale: `print( 1 N)`

## Usare JES

- a volte la matematica può essere sorprendente

- ... provate

```
>>> print 1.0/2.0
```

```
>>> print 1/2
```

- se usate solo **interi** (numeri senza il punto decimale), Python interpreta ciò come intenzione di avere a che fare solo con interi
  - attenzione: le cose sono un po' diverse in Python 3.x

- provate nuove espressioni

```
>>> x = 'Anne'
```

```
>>> y = 'Bob'
```

che cosa vi aspettate che succeda ?

```
>>> print x/2
```

```
>>> print x*y
```

# “Tipo di dato”

- *tipo di dato*: un concetto fondamentale nell'informatica
- elementi costitutivi :
  - **valori**: interi ( 3, -47, 105, ...)
  - reali     (0.482, -21.004, 5.0, ...)
  - stringhe ('bbbb', 'Mark', 'A door', ...)
  - ...
  - **operatori**: aritmetici (+, -, \*, %, ...)
  - per stringhe (+, \*, count, ...)
  - ...
- *valori e operatori* sono connessi gli uni agli altri dal concetto di **tipo di dato**
  - in termini matematici, un *tipo di dato* è correlato al concetto di *struttura algebrica*

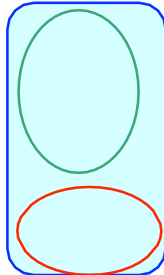
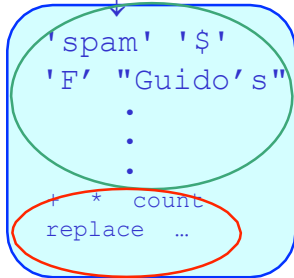
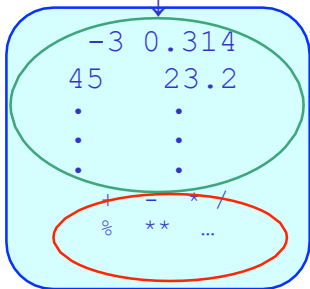


## Tipi di Dato in Python

numeri (int, float)

string (str)

(e altri...)



valori

operatori

**insieme di valori + insieme di operatori *primitivi* = tipo di dato**

# Tipi di Dato in Python

## ■ definizione funzionale degli operatori

■ forma generale:  $op : dom1 \times dom2 \times \dots \rightarrow dom$

- alcuni di questi operatori sono espressi in notazione “infissa”, altri in notazione “funzionale”

## ■ numeri (`int` e `float`)

- $+$  :  $num \times num \rightarrow num$
- $-$  :  $num \times num \rightarrow num$
- $*$  :  $num \times num \rightarrow num$
- $/$  :  $num \times num \rightarrow num$
- $**$  :  $num \times num \rightarrow num$
- $\%$  :  $num \times num \rightarrow num$
- `abs` :  $num \rightarrow num$
- ...

## Tipi di Dato in Python

- definizione funzionale degli operatori

- forma generale:  $op : dom1 \times dom2 \times \dots \rightarrow dom$

- alcuni di questi operatori sono espressi in notazione “infissa”, altri in notazione “funzionale”

- string (`str`)

- $+$  :  $string \times string \rightarrow string$

- $*$  :  $string \times int \rightarrow string$  oppure  $*$  :  $int \times string \rightarrow string$

- `count` :  $string \times string \rightarrow int$

- `find` :  $string \times string \rightarrow int$

- ...

## Strong typing / Static typing

- altri linguaggi di programmazione adottano : *strong typing* + **static typing**
  - p.es.: Java, C++, ...



*dichiarazione di variabili in Java, C++, ...*

```
int myNumber, x;      myNumber e x sono dichiarati come variabili di tipo int
string alfa;          alfa è dichiarato come variabile di tipo string
...
```

**consentito in Java, C++, ...**

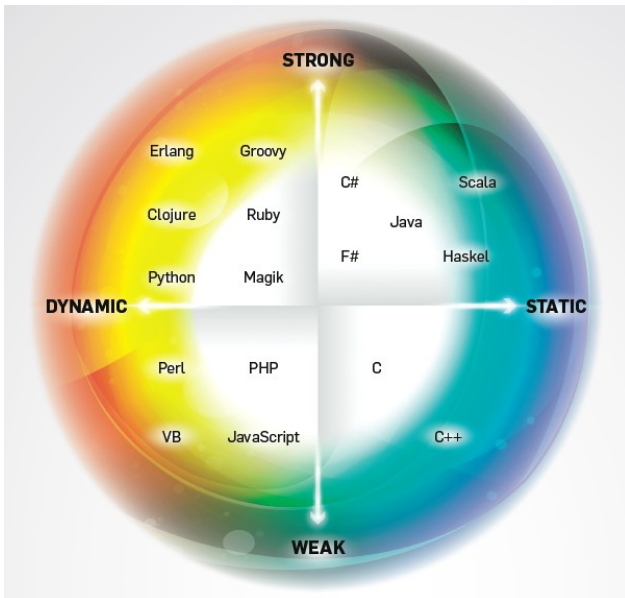
```
...
alfa = "abcd";
x = 3;
myNumber = x * 47;
alfa = "bb" + alfa;
...
```

**vietato in Java, C++, ...**

```
...
alfa = alfa/4;        (1) strong typing
x = alfa;             (2) static typing
myNumber = "aabb" + alfa; (3) static typing
...
```

**invece, il *dynamic typing* di Python  
consente (2) e (3)**

## Panoramica su: *strong/weak - static/dynamic typing*



?



etc. ...

## Iniziamo a lavorare con *multimedia*

- Multimedia: immagini, suoni

- 1° passo:

- come visualizzare un'immagine in JES
- come riprodurre un suono in JES

- 2° passo

- come manipolare immagini e suoni

- 3° passo

- ...

## Visualizzare un'immagine in JES

- di solito, le immagini sono GROSSI oggetti

- ... e non vi farebbe piacere perderle



- memorizzate come *file* in *secondary memory*

- File : un insieme di *byte*, con un *name*

- ancora un altro esempio di coppia [*nome*, *valore*]

- *base name* e *full name*

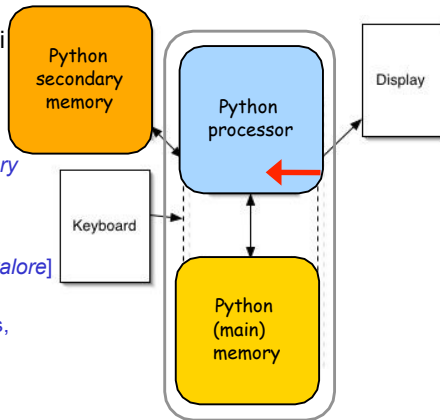
- gestiti dal *sistema operativo*, p.es. Windows, MacOS, Linux

- gestione del *file system*

- I file hanno un *tipo*, di solito indicato con una *estensione* di tre lettere (attaccata al *base name*)

- file .jpg sono di tipo JPEG (immagini), file .wav sono di tipo WAV (suoni)

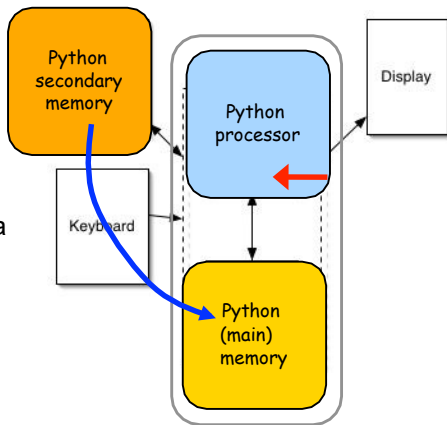
- il tipo del file indica come l'insieme di byte deve essere *interpretato*





## Visualizzare un'immagine in JES

- i file DEVONO essere portati nella *main memory* per poter lavorare con essi
- il loro nome in *secondary memory* è la “maniglia” per trasportarli in *main memory*
  - un (temporaneo) *nuovo nome* può essere dato mentre sono in *main memory*
    - per mezzo del comando “=”  
– *name = picture\_object...*

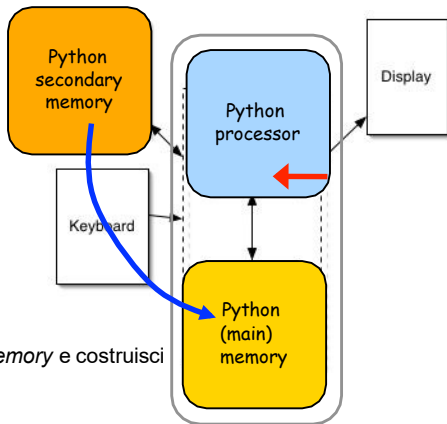


## Visualizzare un'immagine in JES

- i file DEVONO essere portati nella *main memory* per poter lavorare con essi

- passi da seguire

1. Individua un file immagine
2. Prendilo
3. Trasporta i byte da quel file alla *main memory* e costruisci un oggetto di tipo: "Picture"
4. Fai con esso tutto ciò che vuoi
  - p.es.: visualizza l'immagine, modificala, ...

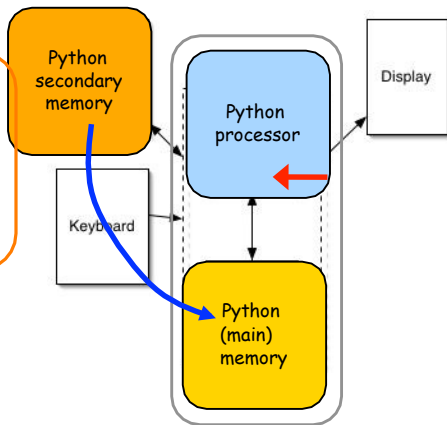


# Visualizzare un'immagine in JES

## ■ passi da seguire = **algoritmo**

1. Individua un file immagine
2. Prendilo
3. Trasporta i byte da quel file alla *main memory* e costruisci un oggetto di tipo: "Picture"
4. Fai con esso tutto ciò che vuoi  
– p.es.: visualizza l'immagine, modificala, ...

## ■ è una procedura *automatica* ?



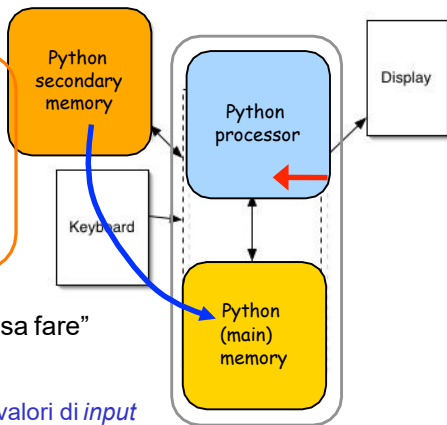
# Visualizzare un'immagine in JES

## ■ passi da seguire = **algoritmo**

1. Individua un file immagine
2. Prendilo
3. Trasporta i byte da quel file alla *main memory* e costruisci un oggetto di tipo: "Picture"
4. Fai con esso tutto ciò che vuoi  
– p.es.: visualizza l'immagine, modificala, ...

## ■ La macchina Python offerta da JES "sa fare" ognuno di questi passi : **funzioni**

- Alcune di queste funzioni accettano valori di *input* (*parametri* della funzione)
- **funzione = nome + codice eseguibile**
  - ancora un altro esempio di coppia [*nome*, *valore*]
  - ... approfondiremo tra poco ...

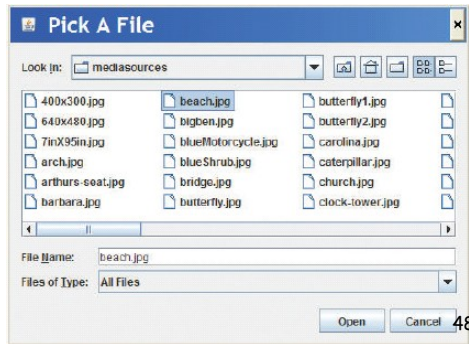


# Visualizzare un'immagine in JES: primo passo

1. Individua un file immagine
2. Prendilo
3. Trasporta i byte da quel file alla *main memory* e costruisci un oggetto di tipo: "picture"
4. fai con esso tutto ciò che vuoi
  - p.es.: visualizza l'immagine, modificala, ...

## ■ funzione pickAFile()

- pickAFile : *none* → *string*
  - funzione **senza parametri**
  - ... attiva il *File Picker*
  - ... restituisce come risultato il *full name* del file selezionato
    - in effetti, una *string*
  - *private*...
- ```
>>> pickAFile()
>>> myFile = pickAFile()
>>> print myFile
```



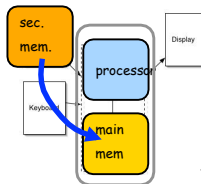
# Visualizzare un'immagine in JES: passi successivi

- 1. Individua un file immagine
- 2. Prendilo
- 3. Trasporta i byte da quel file alla *main memory* e costruisci un oggetto di tipo: "picture"
- 4. fai con esso tutto ciò che vuoi
  - p.es.: visualizza l'immagine, modificala, ...

## ■ funzione makePicture()

- **makePicture** : *string* → *picture*
- funzione con **un parametro**
  - una string che rappresenta un *full filename*
- ... crea e restituisce come risultato un oggetto di tipo *picture*, dal file JPEG identificato da *filename*
  - interpretazione di una sequenza "grezza" di byte
- **provate ...**

```
>>> makePicture(myFile )  
>>> myPict = makePicture(pickAFile())  
>>> print myPict
```



# Visualizzare un'immagine in JES: passi successivi

- 1. Individua un file immagine
- 2. Prendilo
- 3. Trasporta i byte da quel file alla *main memory* e costruisci un oggetto di tipo: "picture"
- 4. fai con esso tutto ciò che vuoi
  - p.es.: visualizza l'immagine, modificala, ...

## ■ funzione show()

■ *show* : *picture* → *none*

■ funzione con un parametro

■ valore di tipo *picture* (espresso tramite un nome, una funzione che lo costruisce, ...)

■ ... visualizza l'immagine in una nuova finestra

■ **non** restituisce un valore (osservate cosa succede nella *command area*)

■ provate ...

```
>>> show(myPict)
```

```
>>> print show(myPict)
```

## ■ altre funzioni per manipolare immagini le vedremo più avanti

■ *getColor()*, *setColor()*, *repaint()*, ... vedi *Help* per il set completo

?

```
>>> filename = pickAFile()
```

Quale è il tipo di dato della variabile `filename` dopo aver eseguito questo comando ?

1) File name

1

2) Picture

2

3) String

3

4) Float

4



# Riprodurre un suono in JES: passi successivi (dopo averlo preso)

## ■ I passi 1. e 2. sono identici al caso delle immagini

- 1. Individua un file suono
- 2. Prendilo

■ 3. Trasporta i byte da quel file alla *main memory* e costruisci un oggetto di tipo "sound"

- 4. fai con esso tutto ciò che vuoi
  - p.e.: riproduci il suono, modificalo, ...

## ■ funzione makeSound()

■ *makeSound* : *string* → *sound*

■ funzione con **un parametro**

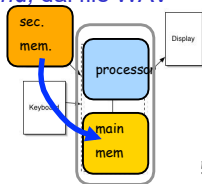
- una string che rappresenta un *full filename*

■ ... crea e restituisce come risultato un oggetto di tipo *sound*, dal file WAV identificato da *filename*

- interpretazione di una sequenza "grezza" di byte

■ *provate ...*

```
>>> myFile = pickAFile()
>>> mySound = makeSound(myFile)
>>> print mySound
```



# Riprodurre un suono in JES: passi successivi

- I passi 1. e 2. sono identici al caso delle immagini

- 1. Individua un file immagine/ suono
- 2. Prendilo
- 3. Trasporta i byte da quel file alla *main memory* e costruisci un oggetto di tipo "sound"
- 4. fai con esso tutto ciò che vuoi
  - p.e.: riproduci il suono, modificalo, ...

- funzione `play()`

- `play : sound → none`

- funzione con **un parametro**

- valore di tipo *sound* (espresso tramite un nome, una funzione che lo costruisce, ...)

- ... **riproduce il suono (ma non aspetta che finisca)**

- `blockingPlay(sound)` invece aspetta che il suono finisca

- **non** restituisce un valore

- **provate...**

- >>> `play(mySound)`

- altre funzioni per manipolare suoni le vedremo più avanti

- `getSample()`, `setSample()`, ... vedi *Help* per il set completo

## Interscambiabilità

- Valori, nomi per quei valori, funzioni che restituiscono quei valori
- nozioni algebriche di **sostituzione** e **valutazione**

```
>>> myFile = pickAFile()
>>> print myFile
```

```
>>> show(makePicture(myFile))
>>> show(makePicture("/Users/imac/Documents/ ... "))
```

} composizione di funzioni !!

```
>>> show(makePicture(pickAFile()))
```

**pickAFile()** *restituisce* un filename (tipo *str*)

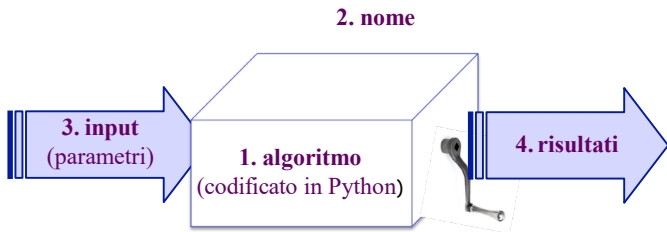
**makePicture()** *riceve* un filename (tipo *str*) e *restituisce* un'immagine (tipo *Picture*)

**show()** *riceve* un'immagine (tipo *Picture*)

**show()** and **play()** invece *non restituiscono* niente,  
(se provate a stamparle usando **print**, ottenete *None*).

## Costruiamo le nostre funzioni

- Funzioni in un linguaggio di programmazione (come Python)



- come specificare e legare tra loro queste 4 cose?

**[*nome, valore*]**

*“codice  
eseguibile”*



**funzione**

## funzione = nome + codice eseguibile (+ parametri)

- **definizione** di funzione

- **sintassi** `def` ( ) :

- : *nome* Python

- : lista di zero, uno, o più *nomi*, separati da virgole:

1, 2, N

- denominati **parametri formali**

- ogni parametro formale è un *nome* Python (possibilmente significativo)

- : *corpo* della funzione

- **indentato** rispetto a `def` (suggerimento: usare (almeno) due spazi)

- **semantica**: definisce una coppia [*nome*, *valore*], dove *nome* è il specificato, e *valore* è il *corpo* ( ) della funzione (➡ codice eseguibile Python)

## funzione = nome + codice eseguibile (+ parametri)

- **chiamata** di funzione

- **sintassi:**

( 1, 2, N )

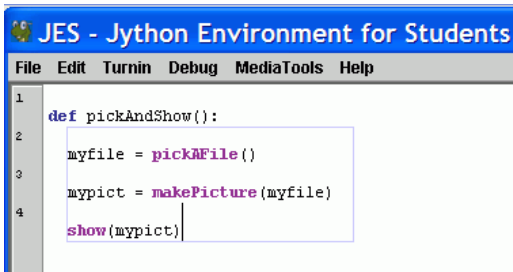
- 1, 2, N è una sequenza di *espressioni*

- **semantica**

- 1. l'interprete Python **valuta** le espressioni 1, 2, N
  - denominate **parametri attuali**
    - ottenendo così una sequenza di valori:  $val_1, val_2, \dots, val_N$
- 2. **associa** ogni parametro formale con il valore dell'espressione corrispondente (per posizione)
  - costruisce un insieme di coppie  $[par_j, val_j]$
- 3. **esegue** il corpo della funzione
  - l'insieme di coppie  $[par_j, val_j]$  esiste solo mentre la funzione è in esecuzione !!

## definizione di funzioni in JES

- Utilizzare la *Program Area*
- JES aiuta a individuare i *blocchi*
  - Comandi indentati nello stesso modo fanno parte dello stesso *blocco*
  - JES racchiude in una cornice blu tutti i comandi che appartengono allo stesso blocco a cui appartiene la linea su cui posizionate il cursore.



```
1 def pickAndShow():
2     myfile = pickAFile()
3     mypict = makePicture(myfile)
4     show(mypict)
```



## Le nostre prime funzioni

- passi da seguire (per visualizzare un'immagine presa dalla memoria secondaria)

- 1. Individua un file immagine (.jpg)
- 2. Prendilo
- 3. Trasporta i byte da quel file alla *main memory* e costruisci un oggetto di tipo: "picture"
- 4. fai con esso tutto ciò che vuoi
  - p.es.: visualizza l'immagine, modificala, ...

procedura (algoritmo) sequenziale

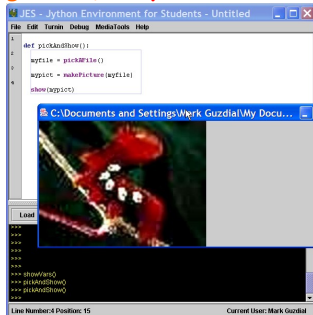
- codifica come funzione Python :

comando per  
associare nome  
a codice

**nome**  
`def pickAndShow ()`

```
myFile = pickAFile()  
myPict = makePicture(myFile)  
show(myPict)
```

codice Python sequenziale



## Le nostre prime funzioni

- passi da seguire (per riprodurre un suono preso dalla memoria secondaria)

- 1. Individua un file suono (.wav)
- 2. Prendilo
- 3. Trasporta i byte da quel file alla *main memory* e costruisci un oggetto di tipo: "sound"
- 4. fai con esso tutto ciò che vuoi
  - p.es.: riproduci il suono, modificalo, ...

procedura (algoritmo) sequenziale

- codifica come funzione Python :

comando per associare nome a codice

```
def pickAndPlay()  
    myFile = pickAFile()  
    mySound = makeSound(myFile)  
    play(mySound)
```

codice Python sequenziale

## un utile aiuto JES

- non vi ricordate più quali coppie [*nome valore*] avete già definito?
  - un nome è libero? è già impegnato?
  - funzione *showVars()*
    - una funzione JES



The screenshot shows a window titled "JES Debug Window #1 - 10:12:43". It contains two tables: "Local Variables" and "Global Variables". Both tables list variables: myfile (String), mypicture (Picture), printNow (Function), and showVars (Function). The values for the functions are truncated. Below the tables is a "Close" button. At the bottom, a terminal window shows the following commands:

```
>>> myfile=pickAFile()
>>> mypicture=makePicture(myfile)
>>> showVars()
>>>
```

## Variante: una funzione per una specifica immagine o suono

- una funzione per una specifica immagine:

```
def showNamed():  
    myfile = "FILENAME"  
    myPict = makePicture(myfile)  
    show(myPict)
```

- una funzione per uno specifico suono:

```
def playNamed():  
    myfile = "FILENAME"  
    mySound = makeSound(myfile)  
    play(mySound)
```

## Variante: una funzione per una immagine o suono *parametrico*

- una funzione per immagini :

```
def showParam(myfile):  
    myPict = makePicture(myfile)  
    show(myPict)
```

- una funzione per suoni :

```
def playParam(myfile):  
    mySound = makeSound(myfile)  
    play(mySound)
```

- che tipo di parametro attuale deve essere utilizzato nella chiamata di queste funzioni ?

## Funzioni con più parametri

- una funzione che riproduce un suono mentre visualizza un'immagine:

```
def playAndShow(sFile, pFile):  
    mySound = makeSound(sFile)  
    myPict = makePicture(pFile)  
    play(mySound)  
    show(myPict)
```

## Funzioni : classificazione

|           |                 | parametri                                                                                                                                |                                                                                                                        |
|-----------|-----------------|------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|
|           |                 | con                                                                                                                                      | senza                                                                                                                  |
| risultato | restituisce     | <code>makePicture()</code> ,<br><code>makeSound()</code>                                                                                 | <code>pickAFile()</code>                                                                                               |
|           | non restituisce | <code>show()</code> ,<br><code>play()</code> ,<br><code>showParam()</code> ,<br><code>playParam()</code> ,<br><code>playAndShow()</code> | <code>pickAndShow()</code> ,<br><code>pickAndPlay()</code> ,<br><code>showNamed()</code> ,<br><code>playNamed()</code> |

- risultato
  - **restituisce**: produce valore utilizzabile “all’interno” della macchina Python
  - **non restituisce**: produce effetto/valore utilizzabile solo “all’esterno” della macchina Python
- come far restituire valori “interni” dalle funzioni che noi definiamo?
  - ... lo vedremo più avanti

## Funzioni : classificazione

|           |                 | parametri                                                                                                                                |                                                                                                                        |
|-----------|-----------------|------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|
|           |                 | con                                                                                                                                      | senza                                                                                                                  |
| risultato | restituisce     | <code>makePicture()</code> ,<br><code>makeSound()</code>                                                                                 | <code>pickAFile()</code>                                                                                               |
|           | non restituisce | <code>show()</code> ,<br><code>play()</code> ,<br><code>showParam()</code> ,<br><code>playParam()</code> ,<br><code>playAndShow()</code> | <code>pickAndShow()</code> ,<br><code>pickAndPlay()</code> ,<br><code>showNamed()</code> ,<br><code>playNamed()</code> |

- parametri : il loro tipo è importante ! (→ *strong typing*)
  - `show(x)` VS. `showParam(x)`
  - `play(x)` VS. `playParam(x)`
  
  - `show: picture → none` vs. `showParam: string → none`
  - `play: sound → none` vs. `playParam: string → none`
    - dove *string* deve essere il *full name* di un file



## Tipo dei parametri: una notazione convenzionale

- **commento** : sequenza di caratteri che inizia con # (in Python )
  - ignorato durante l'esecuzione

```
def playAndShow(sFile, pFile):  
# questo è un commento  
    mySound = makeSound(sFile) # questo è un altro commento  
# e anche questo  
    myPict = makePicture(pFile)  
    play(mySound)  
    show(myPict)
```

- utile per inserire note esplicative
  - sempre consigliabile
  - qui, definiamo una convenzione per dare informazione sul tipo dei parametri

# Tipo dei parametri: una notazione convenzionale

- ispirata allo standard *Javadoc*

```
def      ( 1, 2, ... N):  
# @param par1: tipo; eventuali commenti  
# @param par2: tipo; eventuali commenti  
...  
# @param parN tipo; eventuali commenti
```

**N.B.:** nell'ambito di questo corso, sarà **OBBLIGATORIO** usare questa notazione

esempio

```
def playAndShow (sFile, pFile):  
# @param sFile: string; full name di un file .wav  
# @param pFile: string; full name di un file .jpg  
    mySound = makeSound(sFile)  
    myPict = makePicture(pFile)  
    play(mySound)  
    show(myPict)
```

- Nota: in linguaggi con *static typing* (Java, C++, ...) il tipo dei parametri è specificato nella dichiarazione della funzione

```
( 1 1, ..., N N)  
{ }
```

```
void playAndShow (string sFile, string pFile)  
{ }
```

## consigli pratici per: definizione di funzioni in JES

- Uno degli errori più comuni in JES: dimenticare di fare *Load* (caricare)
  - Una funzione **NON** esiste per JES finchè non è stata *caricata* (*loaded*)
    - prima di essere *caricata*, la tua funzione non è altro che un insieme di caratteri
    - l'operazione di *loading* lo trasforma (codifica) in una funzione eseguibile
  - **Save e Save As**
    - devi fare *Save* prima di *Load*
    - devi fare *Load* prima di poter usare la tua funzione

```
1 def pickAndShow():
2     filename = pickAFile()
3     mypicture = makePicture(filename)
4     show(mypicture)
```

Load Program UNLOADED Watcher Stop

```
>>> pickAndShow()
The error was:pickAndShow
Name not found globally.
A local or global name could not be found. You need to define the function or variable before you try to use it in any way.
>>> |
```

Explain <click> Line Number:4 Position: 18

**Una funzione  
“unloaded” non esiste**

## consigli pratici per: definizione di funzioni in JES

- per facilitarsi la vita
- suggerimento:
  - usa la *command area* per provare singoli comandi
    - verifica di correttezza
  - utilizza il comando `def` nella *program area*
  - fai *copy-paste* nella *program area* di comandi verificati nella *command area*

## In caso di errore ...

- Hai usato *esattamente* gli stessi nomi (maiuscole/minuscole, ortografia)?
- Tutte le righe di un blocco devono essere *indentate*, e *indentate della stessa quantità*.
- Variabili *nominate* in una funzione sono diverse da variabili *nominate* nella *command area*.

- **provate a definire questa semplice funzione:**

```
def foo():  
    x = 10
```

- **e poi, nella *command area* :**

```
>>> x = 20
```

```
>>> foo()
```

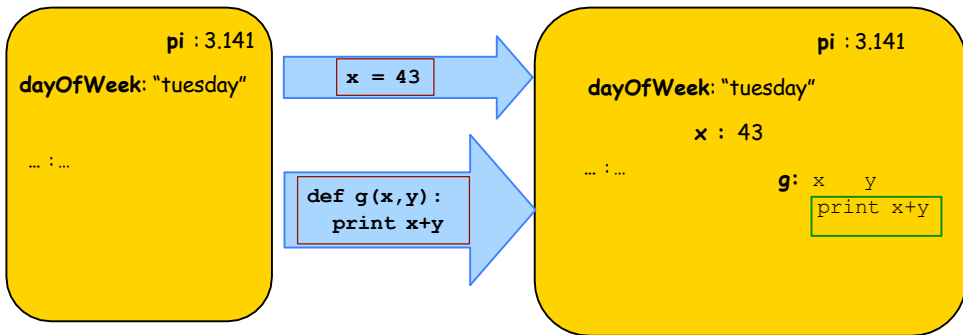
```
>>> print x che cosa vi aspettate che succeda qui ?
```

- **approfondiremo questo argomento nel seguito ...**

- Il computer non può leggere nella tua mente.
  - **fa solo ed esattamente quello che gli dici *esplicitamente* di fare.**

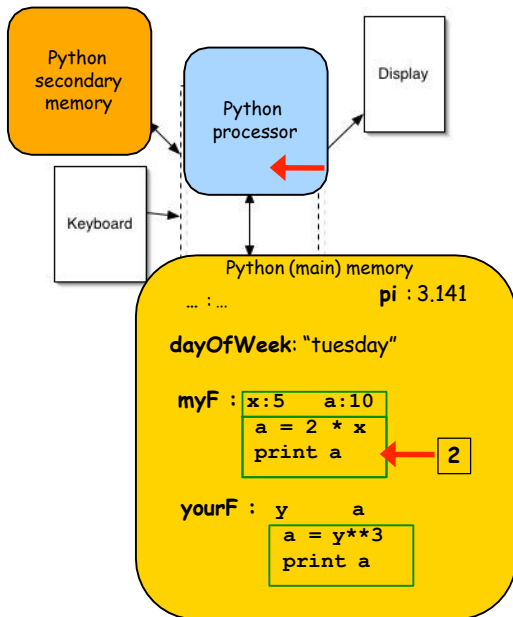
## Ricapitolando

- due modi diversi per creare coppie [nome, valore] nella *main memory*
  - operatore "=" : *valore* è: un dato
  - operatore "def" : *valore* è: un codice eseguibile



## Ricapitolando

- il *processor* gestisce un *indicatore di istruzione corrente*
  - aggiornato in base a regole appropriate
  - per ora: **regola sequenziale**



# Che cosa abbiamo imparato

## ■ Obiettivi di apprendimento

### ■ generali

- utilizzazione dell'ambiente JES per lavorare con Python
- introduzione ai *tipi di dato*
- creare e usare *variabili*, per conservare valori e oggetti
- creare e usare *funzioni*
- scrivere procedure *sequenziali*, e codificarle in funzioni

### ■ casi di studio

- visualizzare immagini
- riprodurre suoni



