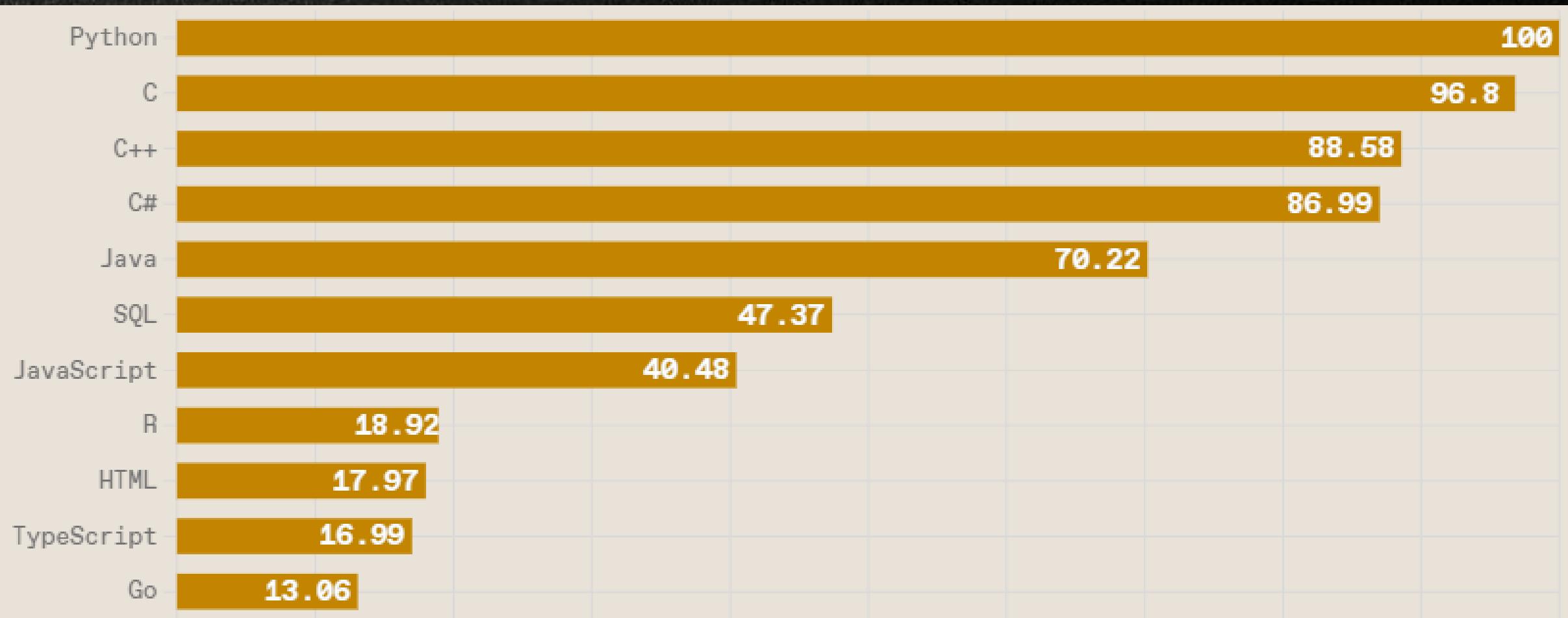


# Python

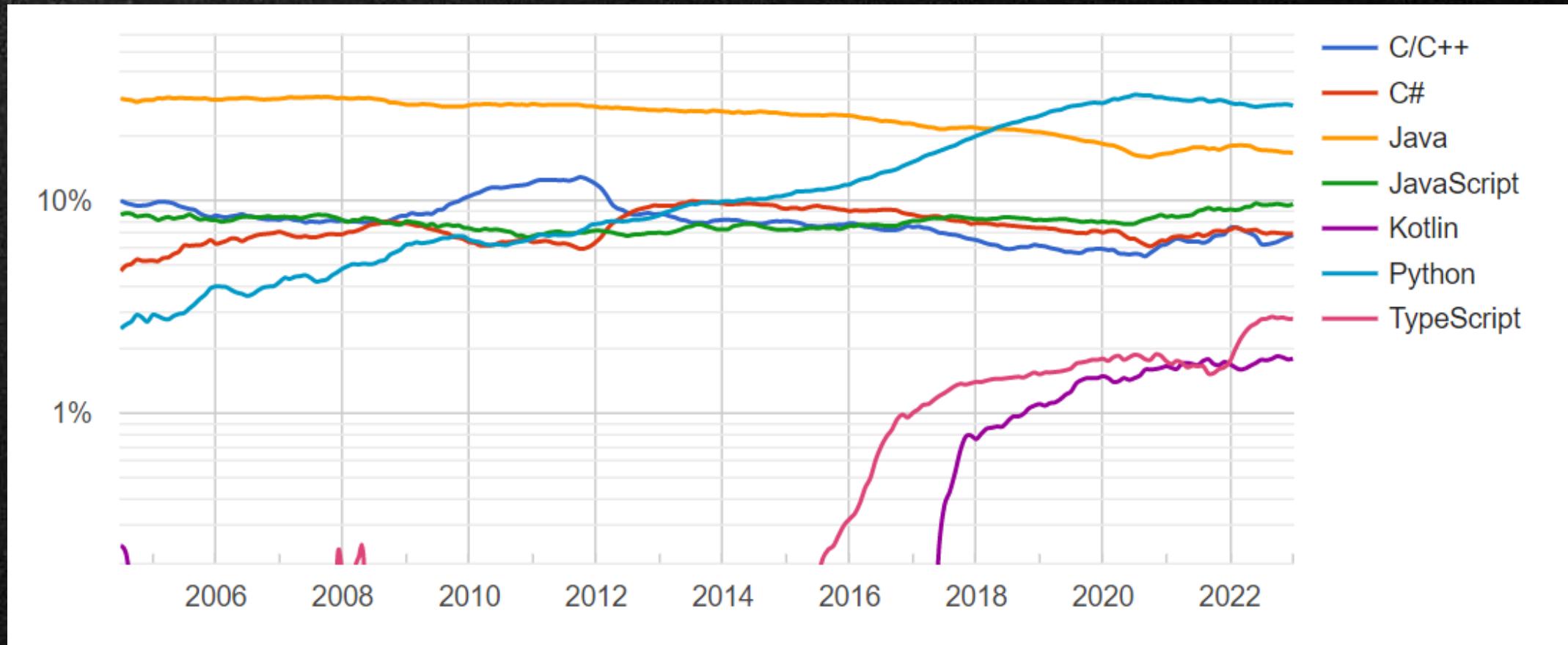
17634 | VISIONE ARTIFICIALE



# IEEE: The Top Programming Languages 2022



# PYPL PopularitY of Programming Language



Fonte: <https://pypl.github.io/PYPL.html> (2023)

# Python

- ▶ Un linguaggio di programmazione di **alto livello general-purpose**
- ▶ Progettato per favorire la **leggibilità** del codice
- ▶ La sua sintassi consente di esprimere concetti con **meno linee di codice** rispetto ad altri linguaggi quali C++ o Java
- ▶ Supporta più paradigmi di programmazione (**orientata agli oggetti**, **imperativa**) e molte caratteristiche della programmazione **funzionale**
- ▶ Dispone di un'ampia libreria standard ed è **facilmente estendibile**
- ▶ Distribuito con licenza **open source** su molteplici piattaforme (Windows, Linux, Raspberry, ...)



# Storia e versioni

- ▶ Ideato da **Guido van Rossum**, che lo inizia a implementare nel 1989. Il nome deriva dalla passione di Guido per il gruppo comico inglese **Monty Python**, attivo principalmente negli anni '70 e '80 (curiosità: il termine informatico 'spam' deriva da un loro sketch).
- ▶ Python **1.0** risale al **1994**: includeva già costrutti per la programmazione funzionale (lambda, map, filter, reduce).
- ▶ Python **2.0** fu rilasciato nel **2000** con molte nuove caratteristiche, fra cui list comprehension, garbage collector e supporto Unicode.
- ▶ Python **3.0** fu rilasciato nel **2008**: si trattò di una grossa revisione del linguaggio non completamente compatibile con le versioni precedenti.
- ▶ Gli esempi di questo corso sono stati sviluppati con Python 3.7



# Visione Artificiale e Python

- ▶ Python si sta diffondendo rapidamente in molteplici settori
- ▶ In **ambito scientifico** e in particolare nel settore dell'**intelligenza artificiale** (e quindi anche della **visione artificiale**) si sta imponendo come il "**prototyping language**" per eccellenza, grazie ai vantaggi che offre:
  - Semplice da imparare
  - Codice facilmente **leggibile**
  - Innumerevoli **librerie** software già pronte per svolgere i compiti più disparati (es. elaborazione immagini, gestione di grandi quantità di dati, generazione di grafici e report, ...)
  - Disponibilità di moduli numerici (scritti in C/C++) molto **efficienti**
  - Gratuito e **open source**
  - Potenti **ambienti di sviluppo** e di **prototipazione rapida** (es. Jupyter notebooks)



# Sintassi

## ▶ Indentazione

- Non solo per la leggibilità del codice, ma definisce i blocchi di codice

## ▶ Commenti

- Iniziano con il carattere '#': il resto della linea è considerato un commento

## ▶ Istruzioni su più linee

- Il carattere '\' permette di dividere una istruzione su più linee
- All'interno di espressioni fra parentesi, si può andare a capo senza il carattere '\'

## ▶ Più istruzioni su una stessa linea

- Mediante il separatore ';'
- Di solito non è una buona idea

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

```
if 42 < 0:
    print("Prova") # Errore
```

```
# L'indentazione definisce i blocchi di codice
if 42 < 0:
    print("Mai stampato 1")
    print("Mai stampato 2")
print("Stampato")
```

```
istruzione_molto_lunga = 3 + 5 + 6 + 7 + 8 \
                           + 9 + 11 + 13
```

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

```
# Più istruzioni su una singola linea
print("Possibile..."); print("ma sconsigliato")
```



# Variabili

- ▶ Create al momento dell'inizializzazione
- ▶ Nomi
  - Case-sensitive
  - Iniziano con una lettera o underscore
  - Caratteri ammessi: Lettere, numeri, underscore (codifica UNICODE)
- ▶ Variabili globali e locali
  - Variabili create fuori da una funzione sono globali, quelle create dentro una funzione sono locali
  - Una variabile creata dentro una funzione con lo stesso nome di una globale è una diversa variabile locale
  - Keyword "global" per creare o modificare una variabile globale dentro una funzione



```
x = 42
y = "Visione Artificiale"
print(x, y)

# Assegnamento dello stesso valore a più variabili
x = y = z = 99
print(x, y, z)

# Assegnamento di più variabili in una linea
x, y, z = "Rosso", "Verde", "Blu"
print(x, y, z)

a = "divertente" # Variabile globale

def funzione():
    a = "facile" # Variabile locale
    print("Python è " + a)

funzione()
print("Python è " + a)
```

# Tipi di dati

- ▶ Il tipo di una variabile è definito al momento dell'assegnamento del valore
- ▶ Tipi di dati predefiniti:
  - Testo: str
  - Numeri: int, float, complex
  - Sequenze: list, tuple, range
  - Dizionari: dict
  - Insiemi: set, frozenset
  - Booleani: bool
  - Binari: bytes, bytearray, memoryview
- ▶ La funzione type() consente di ottenere il tipo di un oggetto

```
x = "Visione Artificiale" # str
print(type(x))

x = 20 # int
print(type(x))

x = 20.5 # float
print(type(x))

x = 1j # complex
print(type(x))
x = ["verde", "rosso", "blu"] # list
print(type(x))
x = ("verde", "rosso", "blu") # tuple
x = range(6) # range
x = {"nome" : "VA", "codice" : 17634} # dict
x = {"verde", "rosso", "blu"} # set
x = frozenset(x) # frozenset
x = True # bool
x = b"ABCD" # bytes
x = bytearray(5) # bytearray
x = memoryview(bytes(5)) # memoryview
```



# Oggetti, valori e tipi

- ▶ Qualunque dato in Python è un *oggetto*
- ▶ Ogni oggetto è caratterizzato da: un *tipo*, un'*identità* e un *valore*
  - Il tipo determina le operazioni che l'oggetto supporta e tutti i suoi possibili valori;
  - L'identità non cambia durante la sua vita;
  - Gli oggetti il cui valore può cambiare sono detti *mutabili*, quelli in cui non può cambiare sono detti *immutabili*.
  - È il tipo a determinare se un oggetto è mutabile o immutabile. Ad esempio numeri, stringhe e tuple sono immutabili, mentre liste e dizionari sono mutabili.
- ▶ Le *variabili* sono *riferimenti* a oggetti

```
# L'assegnamento crea un oggetto in memoria di
# tipo Float, a cui la variabile fa riferimento
risposta = 3.14

print('Tipo:', type(risposta))
print('Identità:', id(risposta))
print('Valore:', risposta)

# L'assegnamento copia il riferimento dell'oggetto
# nella nuova variabile
spam = risposta
print('Stessa identità:', id(spam))

# 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 'risposta' è cambiata,
# mentre l'identità di 'spam' è la stessa
risposta *= 2

print('Identità nuovo oggetto:', id(risposta))
print('Identità vecchio oggetto:', id(spam))
```



# Alcuni oggetti predefiniti

## ► None

- Utilizzato per indicare l'assenza di un valore (simile al `null` di altri linguaggi). `None` è l'unico oggetto della classe `NoneType`

## ► Ellipsis

- Unico oggetto della classe `ellipsis`: può essere indicato con la sintassi `...` (**tre punti consecutivi**). Ne vedremo l'utilizzo in NumPy.

## ► NotImplemented

- Unico oggetto della classe `NotImplementedType`: metodi numerici e di confronto possono restituire questo valore se non implementano l'operazione per determinati operandi.



```
x = None
print(x, type(x))

ret = print('Hello')
# Una funzione senza valore di ritorno
# restituisce None
print(ret)

# None è un oggetto: può essere utilizzato in
# qualsiasi contesto
t = ('Tupla', 'con', 1, 'valore', None)
print(t)

# Sintassi per oggetto Ellipsis
x = ...
y = Ellipsis
print(x, type(x), y, type(y))

# Confronto fra una stringa e un numero:
# restituisce NotImplemented
ret = 'testo'.__eq__(42)
print(ret, type(ret))
```

# Numeri

## ► Tipi numerici predefiniti

- **int**: numero intero, positivo o negativo, di **qualsiasi grandezza**. Attenzione: non è il tipo "int" o "long" che troviamo in genere in altri linguaggi rappresentato con 32 o 64 bit.
- **float**: numeri con la virgola in doppia precisione a livello macchina (le **caratteristiche dipendono dall'architettura** sottostante, di norma formato IEEE-754 a doppia precisione)
- **complex**: numeri complessi rappresentati da una coppia di float.

## ► Conversione fra tipi numerici:

- Possibile utilizzando i relativi costruttori **int()**, **float()**, **complex()**

```
x, y, z = 1, -1, 1.0
print(type(x), type(y), type(z))

x, y, z = 35e3, 12E4, -87.7e100
print(type(x), type(y), type(z))

x, y, z = 3+5j, 5j, -5j
print(type(x), type(y), type(z))

x = 1 # int
y = 2.8 # float
z = 1j # complex

a = float(x) # converte da int a float
b = int(y) # converte da float a int
c = complex(x) # converte da int a complex

print(a, b, c)
print(type(a), type(b), type(c))
```



# Stringhe

- ▶ Sequenza di caratteri **UNICODE**
- ▶ Apici doppi ("...") o singoli ('...')
- ▶ Multi-linea con apici tripli ("""...""")
- ▶ **Non esiste il tipo 'carattere'**: i caratteri sono stringhe di lunghezza uno
- ▶ È possibile accedere alle stringhe come se fossero array (liste Python)
- ▶ Concatenazione stringhe con '+'
- ▶ Funzione predefinita **len()**
- ▶ Ampia collezione di **metodi** (es. **.lower()**)
- ▶ Formatted string literals: f"..."{...}..."

```
print("Sono una stringa")
x = 'Altra'
y = 'stringa'
v = x + ' ' + y # Concatenazione di stringhe

# Stringa multi-linea
soldati = """Si sta come
d'autunno
sugli alberi
le foglie."""
print(soldati)

print(v[0]) # Stringa come 'array'
print(len(v)) # Lunghezza di una stringa
print(v.upper()) # Metodo della classe stringa
print(v.replace("A", "U")) # Altro metodo

x = 34.23322
# Esempio di Formatted string literal
s = f'Formatto {x} con 2 decimali: {x:.2f}'
print(s)
```



# Valori Booleani

- ▶ Due possibili valori: **True**, **False**
  - Sono gli unici due oggetti esistenti della classe **bool**
- ▶ Le espressioni di confronto in Python restituiscono un valore booleano
- ▶ Qualunque **variabile** può essere **convertita** in valore booleano con **bool()**
  - Numeri (**False** se 0, altrimenti **True**)
  - Stringhe (**False** se vuote, altrimenti **True**)
  - Strutture dati come liste, tuple, etc. (**False** se vuote, altrimenti **True**)

```
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({}))
```



# Operatori aritmetici

Operatore	Nome	Esempio
+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	$x / y$
%	Modulus	$x \% y$
**	Exponentiation	$x ** y$
//	Floor division	$x // y$



# Operatori di assegnamento

Operatore	Esempio	Effetto
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
=	x  = 3	x = x   3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

# Operatori di confronto

Operatore	Nome	Esempio
<code>==</code>	Equal	$x == y$
<code>!=</code>	Not equal	$x != y$
<code>&gt;</code>	Greater than	$x > y$
<code>&lt;</code>	Less than	$x < y$
<code>&gt;=</code>	Greater than or equal to	$x >= y$
<code>&lt;=</code>	Less than or equal to	$x <= y$



# Operatori logici

Operatore	Descrizione	Esempio
and	Returns True if both statements are true	$x < 5 \text{ and } x < 10$
or	Returns True if one of the statements is true	$x < 5 \text{ or } x < 4$
not	Reverse the result, returns False if the result is True	<code>not(x &lt; 5 and x &lt; 10)</code>



# Operatori d'identità

Operatore	Descrizione	Esempio
is	Returns True if both variables refer to the same object	x is y
is not	Returns True if the variables refer to different objects	x is not y



# Operatori di appartenenza

Operatore	Descrizione	Esempio
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y



# Operatori bit-a-bit (bitwise)

Operatore	Nome	Esempio
&	AND	Sets each bit to 1 if both bits are 1
	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off



# Condizioni if..elif..else

## ► Esecuzione condizionale di codice

- Notare la sintassi con il carattere ':'
- Come sempre i blocchi di codice sono definiti dall'indentazione
- Eventuale keyword "elif" per specificare ulteriori condizioni con relativi blocchi di codice da eseguire
- Eventuale keyword "else" per specificare un blocco di codice da eseguire se nessuna delle condizioni precedenti è vera
- L'utilizzo di elif multipli permette di ottenere un effetto simile al costrutto "switch...case" di altri linguaggi

## ► Espressione condizionale: forma compatta simile al costrutto "... ? ... : ..." di altri linguaggi

```
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")
```



# Ciclo while e ciclo for

## ► while

- Esecuzione finché condizione True

## ► for ... in

- Iterazione su una sequenza di valori
- range() permette di iterare su una sequenza di numeri

## ► break

- Interrompe il ciclo

## ► continue

- Interrompe l'iterazione corrente e passa alla successiva

## ► else

- Blocco di codice eseguito alla fine del ciclo (anche se non ci sono iterazioni) a meno che non si interrompa con "break"

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

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)
```



# Liste

- ▶ Elenco **ordinato** e **modificabile**, può contenere duplicati
- ▶ Creazione:
  - Sintassi con virgole e **parentesi quadre**
  - Costruttore **list()**
  - **List comprehension**
- ▶ Accesso agli elementi:
  - Con indice fra parentesi quadre (0-based)
  - Possibile usare indice negativo e intervallo di indici (**slicing**)
  - Iterazione con **for...in**
- ▶ Altre operazioni:
  - Funzione **len()**, concatenazione con **"+"**, moltiplicazione con **"\*"**
  - Controllo appartenenza con **"in"**
  - Metodi **append()**, **insert()**, **remove()**, **copy()**, **clear()**, **extend()**, **sort()**, ...

```

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)

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

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

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

# Esempi di "list comprehension"
coloriA = [c for c in colori if c[0]=='A']
quadrati = [x*x for x in range(1,10)]
print(coloriA, quadrati)

```



# Tuple

- ▶ Elenco ordinato e non modificabile, può contenere duplicati
- ▶ Creazione:
  - Sintassi con virgolette e parentesi tonde
  - Costruttore tuple()
- ▶ Accesso agli elementi:
  - Con indice fra parentesi quadre (0-based)
  - Possibile usare indice negativo e intervallo di indici (slicing)
  - Iterazione con for...in
- ▶ Altre operazioni:
  - Funzione len(), concatenazione con "+", moltiplicazione con "\*"
  - Controllo appartenenza con "in"
  - Metodi count(), index()

```

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 è 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)

```



# Slicing

## ► Sintassi $a[i:j]$

- Seleziona gli elementi di "a" con indice  $k$  tale che  $i \leq k < j$  (N.B.  $j$  è escluso)
- Si applica alle strutture dati che contengono una sequenza numerata di elementi (es. liste, tuple, stringhe)
- Se un indice è negativo, gli viene sommato  $\text{len}(a)$ : equivale a contare a ritroso dall'ultimo elemento
- Se  $i$  non è indicato si assume 0
- Se  $j$  non è indicato si assume  $\text{len}(a)$
- Può essere utilizzata anche a sinistra dell'assegnamento, per sostituire o modificare una sotto-sequenza

## ► Sintassi estesa con "step": $a[i:j:s]$

- $k = i + s \cdot n$ , con  $n \geq 0$  e  $i \leq k < j$

```
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]
a[:3] = a[3:]
print(a) # [4, 5, 6, 7, 8, 9, 4, 5, 6, 7, 8, 9]
a[::-2] = [0]*6
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]
```



# Insiemi

- ▶ Collezione **non ordinata** e **non indicizzata**:
  - non può contenere duplicati
  - possono essere aggiunti o rimossi elementi (ma non modificati)
- ▶ Creazione:
  - Sintassi con virgole e **parentesi graffe**
  - Costruttore **set()**
  - **Set comprehension**
- ▶ Accesso agli elementi:
  - Iterazione con **for...in**
- ▶ Altre operazioni:
  - Funzione **len()**, unione con **|**, intersezione con **&**, ...
  - Controllo appartenenza con **"in"**
  - Metodi **add()**, **discard()**, **remove()**, **copy()**, **clear()**, **union()**, **difference()**, **intersection()**, **update()**, ...

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

for c in colori:
    print(c)

colori.remove('Verde')
colori.discard('Giallo') # nessun errore
colori.add('Azzurro')
colori.update({'Arancione', 'Viola'})
print(colori)

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



# Dizionari

- ▶ Collezione **non ordinata** e **modificabile** di coppie **chiave=valore**, non può contenere chiavi duplicate
- ▶ Creazione:
  - Sintassi con virgolette e **parentesi graffe**
  - Costruttore **dict()**
  - **Dictionary comprehension**
- ▶ Accesso agli elementi:
  - Con chiave fra parentesi quadre
  - Metodo **get()**
  - Iterazione con **for...in** sulle chiavi
- ▶ Altre operazioni:
  - Funzione **len()**
  - Controllo presenza di una chiave con "**in**"
  - Metodi **items()**, **keys()**, **values()**, **copy()**, **clear()**, **update()**, **pop()**, ...

```

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():
    print(mat, nome)
for nome in studenti.values():
    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)

```



# Funzioni

29

## ► Definizione di una funzione

- Parola chiave "def" seguita da una o più righe di codice (attenzione sempre all'indentazione)
- Una eventuale stringa come prima istruzione funge da documentazione (**docstring**)

## ► Chiamata di una funzione

- Nome funzione seguito da parentesi tonde

## ► Passaggio dei parametri

- Come qualsiasi variabile Python, i parametri sono **riferimenti a oggetti**.
- Per gli oggetti immutabili (es. numeri, stringhe) l'effetto è sostanzialmente analogo al passaggio per valore.
- Per quelli mutabili (es. liste) una funzione può modificare il contenuto dell'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 è la docstring di calcola"""
    return x * y
print(f"Il risultato è {calcola(6, 7)}.")

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

# Versione più "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)
```



# Parametri delle funzioni

30

- ▶ È possibile specificare **valori di default** per uno o più parametri (gli ultimi), che possono non essere specificati al momento della chiamata
- ▶ Al momento della chiamata è possibile specificare i parametri con **<nome>=<valore>**, in qualsiasi ordine
- ▶ Numero di parametri variabili:
  - **\*args** consente di ottenere una tupla contenente i valori di eventuali argomenti aggiuntivi passati alla funzione
  - **\*\*kwargs** consente di ottenere un dizionario con eventuali argomenti aggiuntivi passati con **<nome>=<valore>**
  - **\*args e \*\*kwargs** possono essere utilizzati singolarmente o insieme

```
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))  
  
def Esame(coro, *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")
```

# Unpacking

## ► Unpacking nel passaggio dei parametri

- L'operatore `*` consente di passare una sequenza (lista, tupla, ...) a una funzione che richiede un elenco di argomenti separati
- L'operatore `**` consente di passare un dizionario a una funzione che richiede un elenco separato di argomenti  
`<nome>=<valore>`

## ► Unpacking nell'assegnamento

- Consente assegnamento multiplo in una sola linea
- L'operatore `*` permette di catturare tutti gli elementi restanti in una lista

```
def calcola(a, b, c):
    return a + b + c

parametri = [4, 18, 20]
print(calcola(*parametri))

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
print(a1, r)

primo, secondo, *altri, ultimo = range(10)
print(primo, secondo, ultimo, altri)
```



# Funzioni come oggetti e lambda

- ▶ In Python anche le funzioni sono oggetti
  - Hanno un tipo, hanno attributi
  - Possono essere utilizzate all'interno di strutture dati (es. liste, dizionari)
  - Possono essere passate come parametri ad altre funzioni
  
- ▶ Funzioni anonime
  - La keyword **lambda** consente di definire semplici funzioni anonime costituite da una singola espressione

```
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))
```



# Alcune funzioni predefinite

- ▶ L'interprete fornisce alcune funzioni che sono sempre disponibili
- ▶ Alcuni esempi li abbiamo già visti:
  - `print()`, `type()`, `id()`, `bool()`, `int()`, `float()`, `str()`, `range()`, `list()`, `tuple()`, `set()`, `dict()`, `len()`
- ▶ Altri esempi di funzioni utili:
  - `enumerate()`: data una sequenza di valori, permette di iterare su una sequenza di tuple del tipo (indice, valore)
  - `zip()`: permette di aggregare più sequenze in un'unica sequenza di tuple
  - `sum()`, `min()`, `max()`: restituiscono rispettivamente la somma, il minimo e il massimo di una sequenza
  - `sorted()`: restituisce una sequenza ordinata

```
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)]
```



## ► Modulo Python:

- File contenente definizioni e istruzioni Python
- Il nome del file è il nome del modulo con estensione .py
- Il comando **import** consente di importare un modulo

## ► Varianti del comando import:

- **import <modulo>**
- **import <modulo> as <nome>**
- **from <modulo> import <n1>, <n2>, ...**
- **from <modulo> import \***
- **from <modulo> import <n> as <n1>**

## ► Variabile **\_\_name\_\_**:

- Stringa contenente il nome del modulo (se importato), oppure "**\_\_main\_\_**" (se eseguito come script)

```
# 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))
```

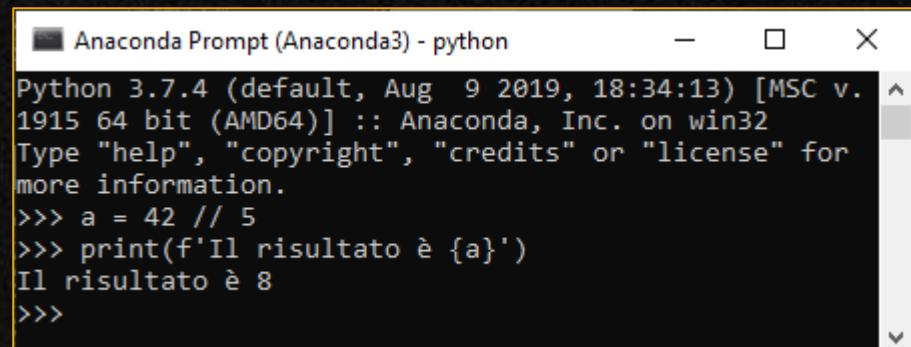
```
from fib import fibonacci
print(fibonacci(90))
```

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

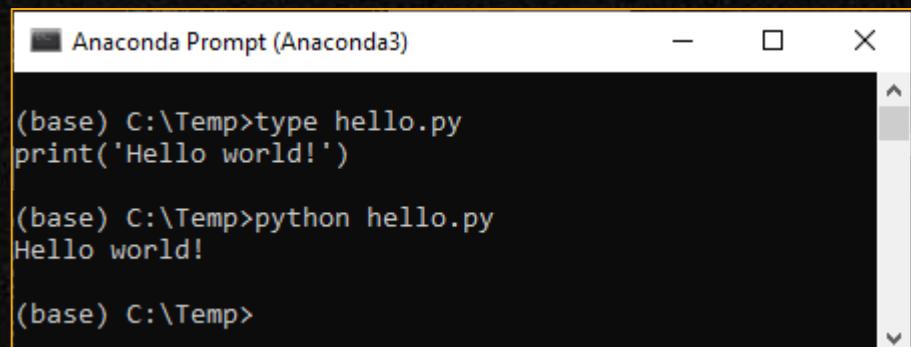
# Interprete

35

- ▶ Per eseguire un programma Python è necessario installare un interprete Python.
  - Esistono versioni dell'interprete per diversi sistemi operativi, scaricabili gratuitamente da <http://www.python.org>.
- ▶ L'interprete può essere utilizzato in modalità interattiva...
- ▶ ...oppure per eseguire un file Python (estensione .py)



```
Anaconda Prompt (Anaconda3) - python
Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v. 1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 42 // 5
>>> print(f'Il risultato è {a}')
Il risultato è 8
>>>
```



```
Anaconda Prompt (Anaconda3)
(base) C:\Temp>type hello.py
print('Hello world!')

(base) C:\Temp>python hello.py
Hello world!

(base) C:\Temp>
```

# Ambiente di sviluppo

- Sono disponibili numerosi IDE per programmare in Python
  - In questo corso all'occorrenza utilizzeremo Visual Studio Code



Screenshot of the Visual Studio Code interface showing a Python development environment.

The interface includes:

- File Bar:** File, Edit, Selection, View, Go, Debug, Terminal, Help.
- Title Bar:** hello.py - Visual Studio Code.
- Search Bar:** SEARCH, Search, Replace.
- Message Bar:** You have not opened or specified a folder. Only open files are currently searched - Open Folder.
- Code Editor:** A file named "hello.py" containing the code: 

```
C: > Temp > hello.py
1 print('Hello world!')
```
- Terminal:** Shows the output of running the script: 

```
PS C:\Users\Raffaele> & C:/Users/Raffaele/Anaconda3/envs/cv/python.exe c:/Temp/hello.py
Hello world!
PS C:\Users\Raffaele>
```
- Status Bar:** Python 3.7.5 64-bit ('cv': conda), 0 △ 0, Ln 2, Col 1, Spaces: 4, UTF-8, CRLF, Python, 1.

# Jupyter notebook

- Applicazione web open-source che consente di creare e condividere documenti che contengono codice eseguibile, equazioni, immagini, grafici e testo esplicativo
  - Sarà utilizzato nelle nostre esercitazioni in laboratorio

**Visione Artificiale**

**Esercitazione: Operazioni sulle immagini**

**Sommario**

- Creazione e modifica di immagini grayscale memorizzate come array NumPy
- Confronto fra velocità di esecuzione di cicli Python e corrispondenti istruzioni NumPy
- Caricamento di immagini da file
- Semplici operazioni su immagini RGB e grayscale
- Binarizzazione di immagini RGB e grayscale: con soglia globale e locale
- Operazioni aritmetiche fra immagini
- Applicazione di una lookup table
- Conversione da RGB a HSL e modifiche ai valori HSL

Iniziamo con l'importazione dei moduli che ci serviranno: `NumPy`, `OpenCV`, `va`. Importiamo anche la funzione `interact` di Jupyter.

```
In [ ]: import numpy as np
         import cv2 as cv
         import va
         from ipywidgets import interact
```

**Esercizio 1** - Nella prossima cella scrivere il codice Python che crea quattro immagini (`immagine1`, `immagine2`, `immagine3`, `immagine4`) con le caratteristiche specificate qui sotto. Eseguire poi anche la cella successiva che contiene alcuni controlli per verificare che le immagini siano state create correttamente.

- `immagine1`: immagine grayscale con dimensione 200x256 (larghezza x altezza), tipo di dato dei pixel `np.uint8`; i pixel di ogni colonna devono contenere, dall'alto verso il basso, tutte le possibili sfumature di grigio [0,255]; tutte le colonne devono essere uguali fra loro;
- `immagine2`: come `immagine1` ma ruotata di 90° in senso antiorario (ovvero la matrice è trasposta);
- `immagine3`: una copia di `immagine1` in cui tutti i pixel con valore multiplo di 3 sono modificati nel valore 0;
- `immagine4`: una copia della sotto-immagine di `immagine1` con origine (pixel in alto a sinistra) alle coordinate (100,150) e dimensioni 20x30.

```
In [ ]: immagine1, immagine2, immagine3, immagine4 = [None] * 4
# --- Svolgimento Esercizio 1: Inizio --- #

# --- Svolgimento Esercizio 1: Fine --- #
va.show((immagine1, 'immagine1'), (immagine2, 'immagine2'), (immagine3, 'immagine3'), (immagine4, 'immagine4'))
```

Ora proseguiamo con alcuni esperimenti su immagini caricate da file. La funzione `cv.imread` carica un'immagine da file e restituisce un array NumPy che ne contiene i pixel. Attenzione: se `if`

“

You primarily write your code to communicate with other coders, and, to a lesser extent, to impose your will on the computer

”

Guido van Rossum - from an Article by Anthony Wing Kosner, November 25, 2019



“

In Python, every symbol you type is  
essential

”

Guido van Rossum - from an Article by Anthony Wing Kosner, November 25, 2019



“

Life is too short to write C++ code

”

David Beazley - EuroScipy 2012 Bruxelles



# The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

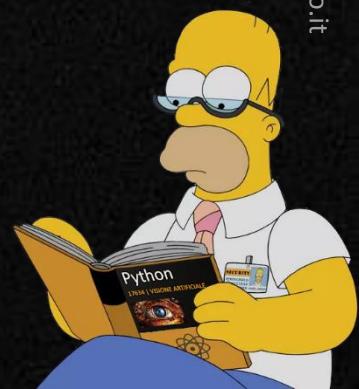
If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!



# Riepilogo

- ▶ In questo corso è necessario prendere confidenza con la sintassi di base di Python:
  - L'**indentazione** (definisce i blocchi di codice)
  - Le **variabili** (definite al momento dell'inizializzazione, contengono **riferimenti** a oggetti)
  - Le principali istruzioni per il controllo di flusso (**if..elif..else**, **for..in**, **while**)
  - I tipi di dati semplici (**int**, **float**, **bool**, **str**) e i principali **operatori**
  - La definizione di **funzioni**, il passaggio dei **parametri**
- ▶ È molto importante comprendere appieno ed esercitarsi con:
  - Liste, **tuple**, **range** e **slicing**
  - Unpacking nell'assegnamento e nel passaggio dei parametri
  - Funzioni con parametri variabili e valori di default
  - Alcune funzioni predefinite come **zip()**, **enumerate()**, **sum()**, **min()**, **max()**



# Per approfondire

- ▶ I siti da cui iniziare:
  - Sito ufficiale: <https://www.python.org>
  - Documentazione: <https://docs.python.org>
  - Comunità italiana: <https://www.python.it>
  - Distribuzione Anaconda: <https://www.anaconda.com>
- ▶ Nota bene: questi lucidi si sono concentrati su alcuni elementi di base di Python che sono indispensabili nel nostro corso; altri aspetti del linguaggio non sono stati affrontati, benché siano molto importanti, come ad esempio:
  - classi ed ereditarietà,
  - eccezioni,
  - coroutine, codice asincrono e concorrente,
  - libreria standard.

