Linguaggi di programmazione

-Python-

Vittorio Bernuzzi, Ph.D Fellow
vittorio.bernuzzi@unipr.it

Cos'è Python e perché impararlo

- Python è un linguaggio di programmazione creato da *Guido van Rossum* negli anni '90.
- È progettato per essere leggibile e comprensibile: la sintassi è pulita e le costruzioni del linguaggio sono orientate alla produttività.
- Per un principiante la leggibilità riduce la barriera d'ingresso;
- Per l'industria la vasta libreria permette di risolvere problemi reali senza reinventare molto codice.

Python nell'industria

Contesti in cui è utile usare Python :

- **Data Science** e **Machine Learning**:
  NumPy, pandas, scikit-learn
- **Sviluppo Web**:
  Django, Flask
- **Automazione** e **scripting**:
  - tool di build, test
- **Prototipazione** rapida:
  sviluppare idea → prova in breve tempo

Differenze principali: Python vs C++

Tipizzazione:

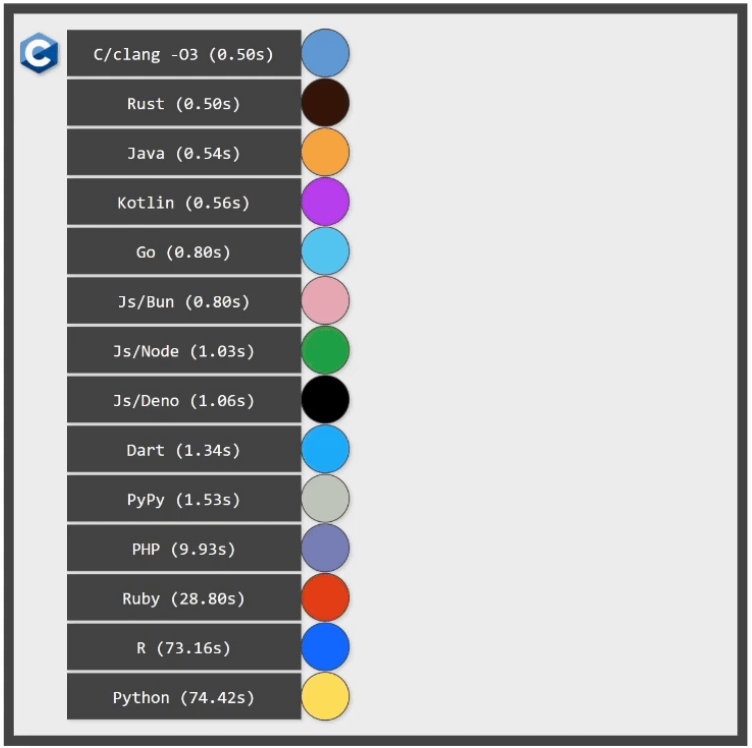
- Python: dinamica, i nomi sono etichette che puntano ad oggetti; i tipi vengono verificati a runtime.
- C++: statica, il tipo è dichiarato e verificato a compile-time.

Gestione memoria:

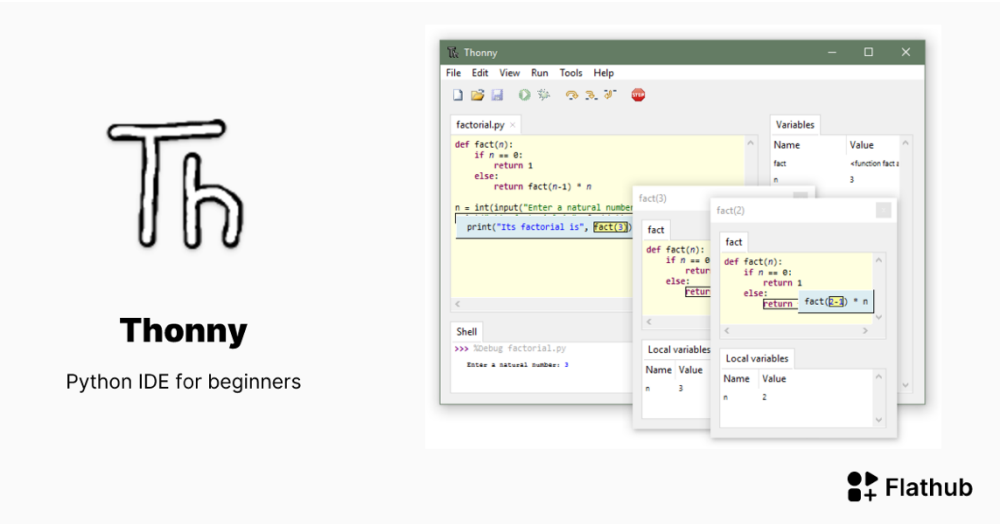
- Python: garbage collection + reference counting.
- C++: gestione manuale (alloc/free) o smart pointers.

Performance e uso:

- C++: migliore per codice molto performante e controllo basso livello.
- Python: migliore per riscrittura rapida, scripting, glue code.



Ambiente consigliato per principianti: Thonny



- Thonny è un'IDE semplice pensato per principianti: ha integrazione con Python, debugger intuitivo e interfaccia pulita.
- Perché scegliere Thonny:
  - Installazione guidata e minima configurazione
  - console integrata e output chiaro
  - Visualizzazione variabili passo-passo durante il debug
- Nasconde complessità non necessarie per iniziare.

Thonny: come installarlo e primo avvio

- Installazione:
  - Windows: scaricare installer dal sito ufficiale Thonny
  - macOS: installer o Homebrew
  - Linux: pacchetti disponibili o uso di pipx
- Primo avvio:
  - Area editor (file), area REPL (Read-Eval-Print Loop), area variabili
  - Eseguire script: Run -> Run current script (F5)

- Consiglio: impostare un progetto per ogni esercitazione (cartella separata) per mantenere ordine tra gli esercizi.

Commentare il codice

- Commentare significa aggiungere spiegazioni leggibili dall'umano ma ignorate dal compilatore/interprete. È fondamentale per rendere il codice comprensibile a sé stessi e agli altri.
    - In **Python** il commento singola riga si scrive con #.
    - In **C++** si usano // per commenti singola riga e /* … */ per blocchi multilinea.

Python

```
# Questo è un commento in Python
x = 10  # Assegno 10 alla variabile x
```

C++

```
// Questo è un commento in C++
int x = 10; /* Commento
su più righe */
```

Consiglio: Commenta il **perché** e non solo il **come**.

Input/Output

- Con operazioni di input/output si intendono i modi in cui i programmi comunicano con l'utente o altri sistemi.

- Comprendere I/O è essenziale perché quasi tutti i programmi leggono dati e producono output.

- Consigli:
  - Validare l'input per evitare crash
  - Fornire messaggi chiari all'utente
  - Separare logica di calcolo da presentazione (stampa) per testabilità

Stampa: `print()`

La funzione `print()` scrive l'output su console.

Parametri utili:

- `end='\n'` imposta cosa aggiungere alla fine (di default newline)
- `file=` per scrivere in file

Esempio:
`print('a','b')`

Formattare l'output aiuta l'utente a leggere meglio i risultati.

Usare f-strings per output leggibile e preciso (Python 3.6+).

```
print(f'Risultato: {3.1415:.2f}')
```

→ print(f'variabile = {variabile}') #variabile = 1

Tipi di variabili base

Intuitivamente, un tipo definisce che cosa puoi fare con un valore (somma, concatenazione, confronto).

Tipi base:
- **int**: numeri interi (contare elementi)
- **float**: numeri con virgola (misure, calcoli)
- **str**: testo
- **bool**: condizioni True/False
- **None**: assenza di valore (utile per default o return vuoto)

Scegliere il tipo corretto semplifica i controlli e previene errori.

```
x = 10 #int

pi = 3.1415 #float

name = 'Luca’ #string

flag = True #bool

no_value = None
```

Lettura da tastiera: `input()` e validazione

`input(prompt)` legge una linea di testo e la restituisce come `str`.

Perché validare:

- L'utente potrebbe inserire dati non attesi (es. lettere invece di numeri)
- Evitare crash con conversioni (int/float)

Esempio pattern:
1) leggere la stringa
2) *cast* al tipo desiderato

```
while True:
    s = input('Inserisci un intero: ')
    try:
        n = int(s)
        break
    except ValueError:
        print('Input non valido, riprova')
```

Casting delle variabili: perché serve e come farlo

Spesso leggiamo stringhe da input ma abbiamo bisogno di numeri per calcoli.

Il '*casting*' converte una variabile da un tipo ad un altro, quando è sensato farlo.

Funzioni principali: int(), float(), str(), bool()

Pro-tip: è importante gestire gli errori di conversione con `try/except` per rendere il programma robusto.

```
s = '123'
n = int(s)  # 123
f = float('3.14')
# gestione errori
try:
    x = int('a')
except ValueError:
    print('Conversione
fallita')
```

Operatori aritmetici: dettagli ed esempi

Operatori standard: +, -, *, /, //, %, **

- `/` restituisce sempre float (motivo: precisione e aspettativa moderna)
- `//` divisione intera per ottenere quoziente (utile per conteggi)
- `%` resto della divisione (utile per controllare parità o cicli)
- `**` elevamento a potenza

```
7 / 3   # 2.3333333333333335
7 // 3  # 2
7 % 3   # 1
2 ** 10 # 1024
```

Esempio: usare // per calcolare quoziente di pagine quando si impaginano risultati.

Best-practice: scegliere l'operatore più espressivo per il problema (evitare hack con float per integer divide).

Divisione e precisione: perché non fidarsi dei float

Curiosità:

I numeri float sono approssimazioni. 0.1 + 0.2 potrebbe non essere esattamente 0.3 a causa della rappresentazione in base-2.

Soluzioni:
- Per visualizzazione usare formattazione (:.2f)
- Per precisione finanziaria usare decimal.Decimal

Esempio:
0.1 + 0.2 -> 0.30000000000000004

Best-practice: per somme monetarie, usare Decimal o lavorare con centesimi (int).

Operatori di confronto e logici: significato pratico

Confronto: ==, !=, <, <=, >, >=

Logici: *and*, *or*, *not*

Usare parentesi per rendere l'ordine delle operazioni esplicito (*da sinistra a destra senza parentesi*).

Esempi:

```
if 0 <= x <= 10:  # molto leggibile
    print(‘in range')
```

```
a = 5
if a > 0 and a < 10:
    print('a in (0,10)')
# chaining
if 0 <= a <= 10:
    print('a in (0,10)')
```

Best-practice: evitare condizioni troppo complesse inline; estrarre in variabili con nome descrittivo.

Identità vs uguaglianza: `is` vs `==`

`==` verifica uguaglianza di valore, `is` verifica se due riferimenti puntano allo stesso oggetto in memoria.

Esempio:

```
a = [1,2]
b = [1,2]
print(a == b)  # True
print(a is b)  # False
```

```
if x is None:
    handle_none()
# non usare x == None
```

Best-practice: usare `is None` e `is not None` per controlli su None.

Struttura IF: sintassi e motivazione dietro la scelta

Python usa indentazione invece di parentesi per delimitare blocchi.

Questo rende il codice più leggibile ma richiede cura negli spazi.

Pattern comune:
- Validazione input
- Branching semplice
- *Clausole di controllo* per casi d'errore

```
if cond:
    # blocco vero
    do_something()
elif other_cond:
    do_other()
else:
    fallback()
```

Best-practice: usare 4 spazi per indentazione e non mescolare tab/spazi.

Formattazione del codice e scope

- Lo *scope* definisce **dove** una variabile è visibile e utilizzabile.
    - In **Python**, lo scope è determinato da funzioni, moduli e classi.
      Non ci sono blocchi {} che creano scope locali (tranne le funzioni).
    - In **C++**, ogni blocco racchiuso da {} crea uno scope:
      le variabili definite dentro un if o un for non esistono all'esterno.

Python

```
x = 10
if (a+b > 20) and (a-b < 5):
    y = 5  # y è visibile anche fuori dal blocco if
print(y)   # stampa 5
```

C++

```
int x = 10;
if (true) {
    int y = 5; // y esiste solo dentro le {}
}
cout << y; // ERRORE: y non definita qui
```

Clausole di controllo e semplificazione delle condizioni

Le clausole di controllo permettono di uscire presto dalla funzione in casi 'speciali', riducendo l’innestamento di blocchi di codice.

Esempio:

```
def process(data):
    if data is None:
        return 'no data'
    # flusso principale
```

Il flusso principale rimane meno indentato e più leggibile.

Esercizi - (https://github.com/Varamyr/formafuturo)

- Es. 1 - Saluto personalizzato
  Leggere il nome dell'utente e stampare 'Ciao <nome>!'.
- Suggerimento: usa input() e f-string
- Es 2 - Somma di tre numeri
  Chiedere 3 numeri all'utente (possono essere float) e stampare la somma e la media.
- Suggerimento: ricorda di convertire le stringhe in numeri con float()
- Es. 3 - Pari o dispari
  Chiedere un intero e stampare se è pari o dispari.
- Suggerimento: usa %
- Es. 4 – Verifica età
  Chiedere l'età e stampare 'puoi guidare' se è uguale o superiore 18, altrimenti 'non puoi guidare'.
- Es. 5 - Calcolo pagine
  Dato il numero di elementi e numero di elementi per pagina, calcolare quante pagine servono.
- Suggerimento: usare divisione intera // e resto della divisione %

VARIABILI COMPLESSE: Liste

Una lista è una collezione *mutabile* di oggetti.
È la struttura più usata per raccogliere dati eterogenei o omogenei.

Perché usare liste:
- Permettono di raggruppare elementi correlati
- Offrono metodi per aggiungere/togliere/ordinare
- Sono intuitive per il principiante (simili a vettori in altri linguaggi)

Imparare le liste è fondamentale per algoritmi di base e manipolazione dati.

Liste: Creazione e accesso agli elementi

Le liste sono una delle strutture dati fondamentali di Python.

Definizione: collezione ordinata e *mutabile* di oggetti (possono contenere tipi diversi).

Creazione:
- Vuota → `lista = []` oppure `lista = list()`
- Con valori iniziali → `numeri = [1, 2, 3, 4]`
- Con funzioni → `range()` in combinazione con
  `list(range(5))` → `[0,1,2,3,4]`

Accesso agli elementi:
- Indici partono da 0.
  `lista[i]` → restituisce l'elemento in posizione `i`.
- Indici negativi → contano dalla fine (-1 = ultimo elemento).

Slicing:
`lista[start:end]`
crea una sottolista dagli indici start fino a end-1.

```
###Inizializzazione

#lista di caratteri
chars = ['a','b',’c’]

#lista di stringhe
spesa = ["pane", "uova", "fagioli"]

#lista di numeri
rainfall_data = [13, 24 , 18, 15]

#lista di 12 elementi uguali a 0
results_by_month =[0] * 12


###Accesso ai valori di una lista

print(lst[0])  # 'a'
print(lst[-1]) # ultimo elemento
print(lst[1:3]) # slice [1,2]
```

Liste: Creazione e accesso agli elementi

- L’indicizzazione a partire da 0 deriva dal calcolo degli offset in memoria: lista[i] indica “spostati di i posizioni dall’inizio”.

- Usare slicing semplifica molto le operazioni di sottoselezione rispetto ai loop manuali.

Consigli:
- Dare nomi significativi alle liste (es. studenti, prezzi).
- Verificare sempre che l’indice sia valido (IndexError se fuori range).
- Usare *slicing* al posto di loop quando il compito è una semplice estrazione di intervalli.

Liste: Creazione e accesso agli elementi

Attenzione ad usare indici validi!

- Lunghezza attuale di una lista “x”: `len(x)`
- Gli elementi sono numerati da `0` a `len(x) - 1`

Una lista può contenere anche altre liste (anche chiamate *matrici*, le vediamo più avanti)

```
spesa = ["pane", "uova", "fagioli"]
n = len(spesa) # 3
spesa[0] # "pane"
spesa[1] # "uova"
spesa[n-1] # "uova"
# sostituisce un valore, len rimane sempre 3
# aggiunge un elemento in coda, la lunghezza aumenta di un'unità
spesa # guess!
```

Liste: gestione elementi (append, pop, insert, remove, sort)

`append(x)`: aggiunge alla fine
`pop(i)`: rimuove l'elemento i-esimo della lista e ritorna elemento
`insert(i, x)`: inserisce alla posizione i
`remove(x)`: rimuove la prima occorrenza
`sort()`: ordina in-place
`index(x)`: trovo l'indice dell'elemento x

Usare metodi adatti risparmia codice e previene bug rispetto a manipolazioni manuali degli indici.

```
nums = [3,1,4]
nums.append(2)  # [3,1,4,2]
nums.sort()     # [1,2,3,4]

#index ritorna l'indice del primo elemento  in argomento
#pop ritorna l'elemento all'indice ritornato da index
x = nums.pop(nums.index(3))
```

Mutabilità: perché conta?

Alcuni oggetti possono essere modificati in-place (liste, dizionari), altri no (int, str, tuple).

In Python, "immutabile" significa che un oggetto non può essere modificato dopo la sua creazione.
Se si cerca di cambiare un oggetto immutabile, non viene modificato l'oggetto originale, ma viene creato un **nuovo** oggetto con i valori desiderati.

Capire la mutabilità aiuta a prevedere effetti collaterali quando si passa una variabile a una funzione.

Esempio pratico:

```
lista = [1,2]
altre = lista
altre.append(3)
# lista è ora [1,2,3]
```

Motivazione: preferire funzioni pure (senza effetti collaterali) quando possibile per facilità di debug.

Variabili complesse: Stringhe

Le stringhe sono sequenze immutabili di caratteri usate per rappresentare testo.

Sono *immutabili* per motivi di sicurezza e hashing.

Alcune operazioni importanti sono:

- **Concatenazione:** unire più stringhe con l'operatore +
- **Ripetizione:** usare l'operatore * con un intero per ripetere una stringa
- **Slicing:** estrarre una parte di stringa usando `[start:end:step]` (alcuni dei parametri possono essere impliciti)

La **concatenazione** è utile per costruire messaggi dinamici.
Lo **slicing** semplifica l'estrazione di sottostringhe senza dover scrivere loop manuali.

Stringhe: esempi operazioni

```
s1 = "Ciao"
s2 = "Mondo”

#Concatenazione
print(s1 + " " + s2)    # "Ciao Mondo”

#Ripetizione
print("ha" * 3)         # "hahaha"

#Slicing
s = "Python"
print(s[0:4])           # "Pyth"
print(s[-3:])           # "hon"
print(s[::-1])          # "nohtyP”

# [start:end:step] = [::-1]
# significa l’intera stringa con step=-1
```

Stringhe: metodi utili (split, join, strip)

Python fornisce diversi metodi integrati per manipolare stringhe in modo semplice.

- `split(sep)`: divide una stringa in una lista, usando come separatore sep (default spazi).
- `join(lista)`: unisce gli elementi di una lista in un'unica stringa, usando il testo su cui è chiamato come separatore.
- `strip()`: rimuove spazi (o caratteri specificati) da inizio e fine della stringa.

Questi metodi sono indispensabili per l'elaborazione di input testuali (es. file, dati da tastiera). Rendono più semplici operazioni comuni senza bisogno di scrivere funzioni manuali.

Consigli:
- Usare `split()` senza argomenti per dividere correttamente anche sequenze multiple di spazi.
- Con `join()`, ricordare che gli elementi della lista devono essere stringhe.
- `strip()` è utile per "pulire" dati letti da file o input dell'utente.

Stringhe: metodi utili (split, join, strip)

```
frase = "Python è divertente"
print(frase.split())              # ['Python', 'è', 'divertente']

lista = ["2025", "09", "06"]
print("-".join(lista))            # "2025-09-06”
print(lista.split(”-"))           # ["2025”, ”09”, ”06"]

spazi = "   ciao   "
print(spazi.strip())              # "ciao"
```

Immutabilità delle stringhe

Come accennato le stringhe, diversamente dalle liste, non possono essere modificate in-place (senza creare una copia dell'oggetto): ogni operazione crea un nuovo oggetto.

Esempio:

```
s = 'abc'
s[0] = 'x'  # errore

lst = ['a','b']
lst[0] = 'x'  # funziona
```

Immutabilità rende gli oggetti *hashable* e sicuri come chiavi di `dict` (altra struttura di dati che vedremo più avanti).

Esercizi su liste e stringhe

Esercizio 1
Scrivi un programma che chieda all'utente di inserire, su una singola riga, una sequenza di numeri separati da virgole (es.: 10, 5, 42). Il programma deve leggere questa riga e stampare a video la lista dei numeri inseriti (come una struttura dati lista).

Esercizio 2
Realizza un programma che richieda all'utente una riga contenente numeri separati da spazi.
Il programma deve leggere quella riga, calcolare la somma di tutti i numeri e stampare il risultato.

Esercizio 3
Scrivi un programma che chieda all'utente di inserire una frase.
Il programma deve stampare a video la lista delle parole contenute nella frase (ogni parola come elemento della lista).

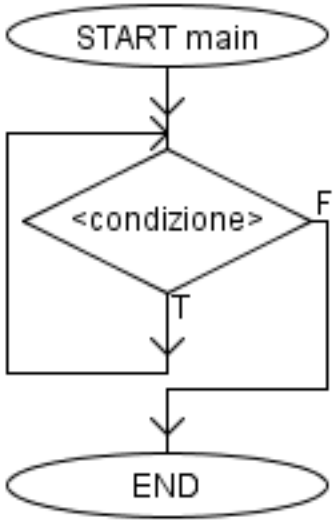
Esercizio 4
Chiedi all'utente di digitare una frase. Il programma deve stampare quante parole ci sono nella frase e quale è la parola più lunga (stampa solo il testo della parola più lunga).

Esercizio 5
Crea un programma che legga dall'utente una lista di elementi (una riga con elementi separati da virgole). Il programma deve produrre e stampare una nuova lista in cui sono stati rimossi i duplicati, mantenendo l'ordine originale di comparsa.

Esercizio 6
Scrivi un programma che chieda all'utente di inserire una frase e poi stampi la stessa frase con l'ordine delle parole invertito (l'ultima parola diventa la prima, e così via).

I Cicli

- I **cicli** (loop) consentono di ripetere automaticamente un blocco di istruzioni più volte.
- Sono fondamentali quando dobbiamo elaborare collezioni di dati (liste, stringhe, file) o ripetere azioni fintantoché una condizione non cambia.
- Evitano la duplicazione di codice e rendono i programmi generali e riutilizzabili.
- Con i cicli possiamo processare input di lunghezza variabile senza riscrivere istruzioni per ogni elemento.



*Consiglio: prima di scrivere un ciclo, pensa a **cosa deve cambiare** ad ogni iterazione (variabile di stato) e qual è la **condizione di terminazione**.*

Cicli: `while`

Il ciclo `while` esegue il corpo finché la condizione booleana è vera.

Si usa quando non si conosce in anticipo il numero di iterazioni (es. leggere input fino a quando l'utente non scrive exit).

Attenzione ai loop infiniti: serve sempre una strategia per far diventare la condizione false; dimenticare `i += 1` *→ loop infinito.*

while è la scelta naturale per processi guidati dallo stato (es. attesa di evento).

Consiglio: assicurati che la variabile aggiornata cambi in direzione di terminazione; preferisci guard clauses per casi d'errore.

```
#Esempio di un ciclo while

i = 0
while i < 5:
    print(i)
    i += 1  # aggiornamento contatore
```

Cicli: `for`

In Python il ciclo `for` itera direttamente su un oggetto *iterabile* (lista, stringa, range, ecc.).

È la forma preferita quando dobbiamo eseguire la stessa operazione per ogni elemento di una sequenza.

Non serve gestire manualmente l'indice (ma può essere ottenuto con `enumerate`).

In certi casi è più vantaggioso rispetto al ciclo `while` perchè è più leggibile e meno soggetto ad errori rispetto ad un contatore manuale.

```
#Esempio di ciclo for

nomi = ["Anna", "Luca", "Marta"]
for nome in nomi:
    print("Ciao", nome)
```

Operazioni utili per cicli: `range()`

`range(start, stop[, step])` genera una sequenza di numeri (lazy).
È comodo per ripetere un'azione un numero preciso di volte o per ottenere indici.

```
for i in range(5):          # 0,1,2,3,4
    print(i)

for i in range(1, 11, 2):   # 1,3,5,7,9
    print(i)
```

Comodo perchè evita la costruzione manuale di liste intermedie; efficiente in memoria.

Consiglio: quando serve l'indice e il valore, preferisci `enumerate()` *(slide successiva)*

Operazioni utili per cicli: `enumerate()`

`enumerate(iterable, start=0)` restituisce coppie `(indice, valore)` ed evita l'uso di `range(len(...))`.
Migliora leggibilità e previene errori di indice.

```
frutti = ['mela', 'pera', 'uva']
for i, frutto in enumerate(frutti, start=1):
    print(i, frutto)

#OUTPUT:
# 1, mela
# 2, pera
# 3, uva

## qui la numerazione parte da 1 perché indicato nel
parametro start
## se non indicato, di default lo start è uguale a 0
```

Operazioni utili per cicli: `break` e `continue`

Ci sono occasioni in cui è necessario interrompere prematuramente un ciclo oppure saltare un iterazione in conseguenza a pre-determinate condizioni che si verificano.

In questi casi è possibile utilizzare:

- `break:` interrompe immediatamente il ciclo (utile per trovare il primo elemento che soddisfa una condizione).
- `continue:` salta il resto del corpo e passa all'iterazione successiva (utile per ignorare casi particolari).

Consiglio: usare con parsimonia, un abuso può rendere il codice difficile da seguire.

```
for x in range(10):
    if x == 5:
        break      # esce dal ciclo completamente
    if x % 2 == 0:
        continue   # salta la stampa se pari
    print(x)

#OUTPUT?
```

Gestire cicli infiniti

Un loop infinito è un ciclo che non termina: può bloccare il programma se non gestito correttamente.

Talvolta sono intenzionali (server che rimangono in ascolto).

Ogniqualvolta si decide di utilizzare un ciclo infinito intenzionalmente è buona norma assicurarsi di definire una condizione di uscita.

```
while True:
    cmd = input(”Inserisci un comando > ")
    if cmd == ”esci":
        break
```

Cicli annidati

I loop possono essere annidati: un loop dentro un altro.

Utile per matrici, tabelle e altre operazioni.

Attenzione: la complessità cresce rapidamente.

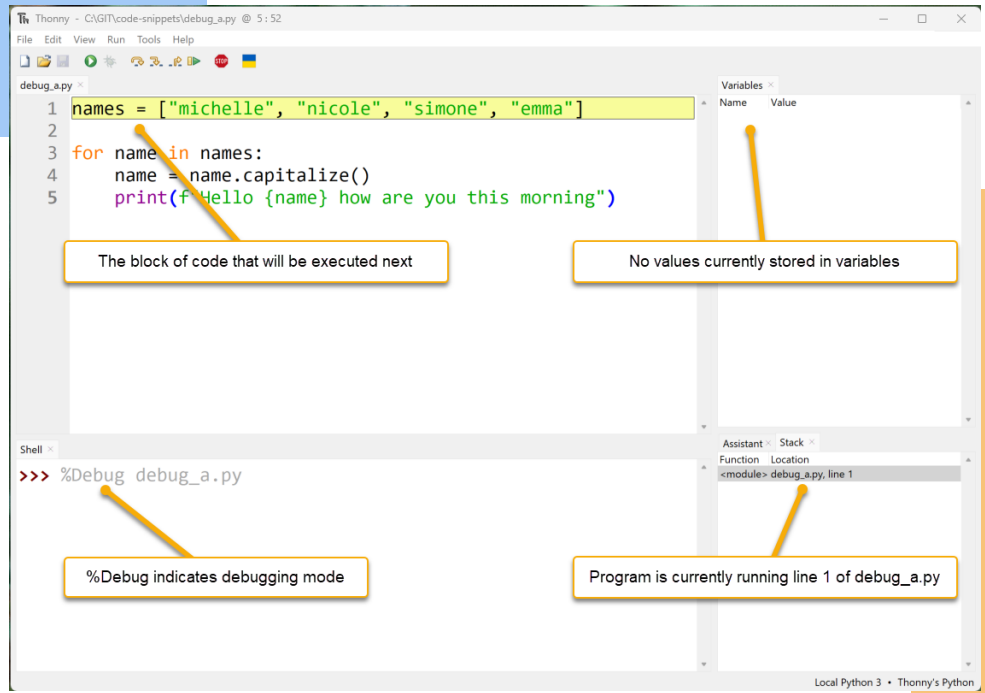
```
matrix = [[1,2],[3,4]]
for i in range(len(matrix)):
    for j in range(len(matrix[i])):
        print(matrix[i][j])
```

Usare cicli annidati solo quando necessario; cercare algoritmi più efficienti se la dimensione cresce.

Debugging dentro ai cicli

Se un loop si comporta male:

- stampa variabili chiave (`print`) per capire l'evoluzione dello stato
- riduci il caso di test (usa input piccolo)
- inserisci `break` temporanei per controllare il comportamento
- utilizza il debugger integrato nell'IDE per svolgere l'esecuzione riga per riga



Esercizi su cicli - parte 1

Esercizio 1 — Conta vocali con `for`
Chiedi all'utente di digitare una frase da tastiera.
Il programma deve leggere la frase e, usando un ciclo `for`, determinare il numero totale di vocali presenti (considera sia maiuscole sia minuscole).
Stampa a video il conteggio finale.

Esercizio 2 — Somma progressiva con `while`
Chiedi all'utente, ripetutamente, di inserire numeri interi da tastiera.
Usa un ciclo `while` per continuare a leggere fino a quando l'utente inserisce il valore 0 (zero).
Alla fine, stampa la somma di tutti i numeri inseriti.

Esercizio 3 — Elenco parole lunghe con `for`
Chiedi all'utente di inserire una frase. Il programma deve creare una lista contenente tutte le parole della frase che hanno più di 4 caratteri, utilizzando un ciclo for.
Alla fine stampa la lista risultante.

Esercizio 4 — Tabellina con `for` e `range`
Chiedi all'utente di inserire un numero intero N. Il programma deve stampare la tabellina di N da 1 a 10, una riga per ciascun prodotto (formato libero leggibile).

Esercizio 5 — Filtrare elementi pari da lista numerica
Chiedi all'utente di inserire sulla stessa riga una sequenza di numeri separati da spazi. Il programma deve costruire una nuova lista contenente solo i numeri pari, usando un ciclo `for`, e stampare la lista.

Esercizi su cicli - parte 2

Esercizio 6 — Primo valore che soddisfa la condizione (`break`)
Chiedi all'utente di inserire una lista di parole (una riga, parole separate da spazio).
Il programma deve trovare e stampare la prima parola che contiene la lettera 'z' (o un messaggio se nessuna parola la contiene).
Usa un `for` e `break` per uscire non appena trovi la parola.

Esercizio 7 — Conta occorrenze e indice con `enumerate`
Chiedi all'utente di inserire una frase e una parola (due input distinti).
Il programma deve cercare quante volte la parola compare nella frase e stampare il conteggio, inoltre deve stampare il primo indice (0-based) della lista delle parole in cui appare (o -1 se non appare).
Usa `split()` per ottenere la lista di parole e enumerate per trovare l'indice.

Esercizio 8 — Inversione parola per parola con `for` e `join`
Chiedi all'utente di inserire una frase.
Il programma deve creare una nuova frase in cui l'ordine delle parole è invertito (ultima parola prima), ricomponendo la stringa finale con join.
Usa un ciclo per creare la lista invertita (non usare lo slicing [::-1] in questo esercizio). Stampa la frase invertita.

Esercizio 9 — Rimuovi elementi dispari dalla lista (modifica in-place sicura)
Chiedi all'utente di inserire numeri separati da spazi. Costruisci una lista iniziale e poi, usando un ciclo for su una copia o usando un approccio sicuro, rimuovi tutti gli elementi dispari dalla lista originale. Alla fine stampa la lista modificata.

Esercizio 10 — Somma condizionale con `for` e `continue`
Chiedi all'utente di inserire una riga di numeri separati da spazi e poi un valore soglia T. Il programma deve sommare solo i numeri strettamente maggiori di T, usando continue per saltare i numeri non validi. Stampa la somma finale.

Esercizi complessivi

Esercizio 1 — Verifica formati dati e aggregazione
Chiedi all'utente di inserire una riga con voti scolastici separati da spazi (numeri interi da 0 a 30).
Il programma deve: verificare (tramite `if`) che ogni valore sia nel range valido; ignorare i valori non validi; quindi stampare la media dei voti validi arrotondata come preferisci.
Se non ci sono voti validi, stampa un messaggio esplicativo.

Esercizio 2 — Estrarre e contare parole chiave da testo
Chiedi all'utente di inserire una frase e una lista di parole chiave separate da virgola (es. errore,warning,info) — entrambe da tastiera.
Il programma deve normalizzare la frase (rimuovere punteggiatura semplice e usare `lower()`), dividere la frase in parole, poi contare quante occorrenze totali delle parole chiave appaiono.
Stampa il conteggio totale e la lista delle parole chiave trovate almeno una volta.

Esercizio 3 — Semplice validatore password e statistiche
Chiedi all'utente di inserire una password; poi chiedi di inserire una lista di password (una riga, separate da virgole) come "blacklist".
Il programma deve verificare, usando `if` e cicli, se la password è presente nella blacklist (stampa Rifiutata), altrimenti deve calcolare e stampare: lunghezza della password, quante lettere maiuscole contiene, quante minuscole e quanti numeri. (Solo operazioni su stringhe e cicli. niente regex.)

Esercizio 4 — Simulazione semplificata di paginazione
Chiedi all'utente il numero totale di elementi N (intero) e quanti elementi vanno in ogni pagina P (intero). Poi chiedi quale pagina k vuole visualizzare (intera, 1-based). Il programma deve calcolare quali indici (0-based) degli elementi ricadono nella pagina k e stampare una lista fittizia degli indici di quegli elementi (se k è fuori range, stampa un messaggio). Usa if, operazioni aritmetiche (//, %), e, se serve, for o range per generare la lista di indici.

Esercizio 5 — Mini-analisi testo con filtro e output formattato
Chiedi all'utente di inserire un paragrafo di testo. Il programma deve: pulire il testo (`strip()`), creare la lista di parole (`split()`), costruire una nuova lista contenente solo le parole di almeno 4 caratteri, poi stampare per ogni parola della nuova lista una riga con indice (a partire da 1) e la parola stessa, formattando l'output in modo leggibile.