Guida all'installazione di Python e del suo ecosistema su Windows 10/11

Gennaio 2026

Sommario

COMPONENTI NECESSARIE AL CORSO

Per il corso ci servono 4 componenti: python ed i package python, miniconda, l'ambiente virtuale python e Visual Studio Code (cioè l'ambiente di sviluppo in Python):

Attenzione: non confondere l'ambiente virtuale Python, che in questo documento indicheremo con **VE** (*virtual environment*) e che è una cartella del PC, con una macchina virtuale, che in questo documento indicheremo per brevità **VM** (*virtual machine*) e che è un insieme di risorse hardware e software (alternativo al proprio PC per chi non possa installarvi software.

Procedere nel modo indicato nei prossimi capitoli, senza modificare l'ordine delle attività.

PYTHON

In generale, i nuovi computer hanno python già installato, in genere non l'ultima versione – NON TOCCARLO!! può dare problemi al sistema operativo!

In Windows 10/11 l'eseguibile Python si chiama ***python.exe*. E' l'interprete python**.

Come possiamo vedere **tutte le installazioni di Python** sul nostro PC? → Digitate in un terminale DOS (attivabile dal menù Windows → ***cmd***) il seguente commando:

where python

Si ottiene una lista delle (eventuali) cartelle dove esiste il file *python.exe* (possono essere 0, 1 o diverse). La lista è estratta, come vedremo anche più avanti, dalla **variabile di ambiente PATH di Windows** (nel suo ordine).

Come controllare la versione di python attiva? Cioè quella che, tra le molte eventualmente installate e prima elencate, **viene eseguita** quando si digita sul terminale: *python*. Posizionarsi su una delle cartelle ottenute dal comando DOS precedente e digitare (sempre dal terminale DOS):

python --version

oppure: ***python3 --version***

In questo documento assumiamo per semplicità che python NON sia installato sul vostro PC o macchina virtuale (VM).
Se invece Python è già installato, non è un problema, useremo comunque la versione Python dell'ambiente virtuale che andremo più avanti a creare, non questa!

COME SI SCRIVE CODICE PYTHON

Ci sono varie possibilità:

- da terminale DOS, o meglio da un **anaconda prompt**, [1] attivare la shell di python con il comando

  python

- e poi eseguire un comando python per volta od uno script python
- da IDLE
- con una **IDE**, come PyCharm o Spyder

[1] Vedi più avanti per la definizione di un anaconda prompt

- con una **interfaccia notebook**, come Visual Studio Code, Jupyter Notebook, Jupyter Lab, Google Colab, che permettono di editare i **notebook**. [2]

Le IDE non sono adatte alla formazione, in primis perché consentono l'edit del solo codice e non anche di testo formattato, immagini, link Internet, come invece è possibile con un'interfaccia notebook.

La soluzione migliore per la formazione e per i progetti esplorativi sono i notebook.

INTERFACCIA NOTEBOOK

Per editare un notebook serve un'interfaccia notebook. Le seguenti – molto diffuse - hanno alcuni **requisiti tecnico-organizzativi**:

- VSCode, Jupyter Notebook, Jupyter Colab: si devono **installare** in locale o su un server
- Google Colab richiede **un account Google** (personale o aziendale).

Per aziende con requisiti elevati di sicurezza, cioè per le quali le 2 precedenti opzioni non sono praticabili, si deve creare **una macchina virtuale** (VM).

INSTALLAZIONE DI PYTHON

Se Python NON è ancora installato sul vostro PC / VM, oppure se è la versione di sistema, occorre installare una **nuova** versione di python, in molti modi (alternativi):

- ad esempio dal sito python ufficiale,
- oppure, meglio, da una distribuzione python come Miniconda o Anaconda.

Scegliamo la seconda possibilità (una distribuzione, che installa al suo interno anche Python), perché il solo Python non è sufficiente, servono anche i **package python**, che arricchiscono python (e la sua libreria standard) con librerie ad hoc per vari scopi. I package python, come vedremo, sono distribuiti attraverso alcuni **canali** di distribuzione.
E' meglio usare **miniconda**. Si sconsiglia di installare Anaconda.

[2] Vedi più avanti per la definizione di notebook

GLI AMBIENTI VIRTUALI E CONDA

Per lavorare in modo pulito con Python e non avere problemi nel tempo, occorre creare un ambiente virtuale (VE=Virtual Environment) python.

Come si crea un VE python ed a cosa serve?

Dalla voce Wikipedia IT "conda":

> *Un ambiente virtuale consente all'utente di creare diversi set di pacchetti software per ogni progetto su cui stanno lavorando. Gli ambienti virtuali sono essenziali per gestire progetti Python con requisiti diversi, garantendo la flessibilità, l'organizzazione e la riproducibilità del proprio lavoro.*
>
> *Ad esempio, supponiamo di avere due progetti con le seguenti caratteristiche:*
>
> - ***Progetto A:** Richiede Python 3.7 e le librerie TensorFlow 1.15 e Keras 2.3.1.*
> - ***Progetto B:** Richiede Python 3.9 e le librerie PyTorch 1.10 e scikit-learn 1.0.*
>
> *È possibile creare due ambienti, uno per il Progetto A e uno per il Progetto B.*
>
> - ***Ambiente A:** Con Python 3.7, TensorFlow 1.15 e Keras 2.3.1.*
> - ***Ambiente B:** Con Python 3.9, PyTorch 1.10 e scikit-learn 1.0.*
>
> *In questo modo è possibile passare da un progetto all'altro semplicemente attivando l'ambiente corrispondente e senza preoccuparsi di conflitti tra le versioni delle librerie.*

conda è il software migliore per **creare e gestire ambienti virtuali in Python** [3]. Nel seguito del documento, come detto, a volte ci riferiremo ad essi, per brevità, con "ambienti (virtuali)" oppure con la sigla VE (**V**irtual **E**nvironment).

---

[3] Ecco **un'altra utile definizione di VE** python: *"A (virtual) "environment" in Python is the CONTEXT in which a Python program runs that consists of **a Python interpreter and any number of installed versioned packages**." A (virtual) environment allows you to install packages **without affecting other environments, isolating your workspace's package installations*** (dal sito ufficiale di VSC). In questo modo possono coesistere **progetti Python differenti, sviluppati in tempi differenti con versioni di Python e dei package tra loro non compatibili** [nota mia]**.**
E' essenziale comprendere che un VE è un oggetto FISICO sul proprio PC (non sta in cloud): **"a virtual environment creates a FOLDER that contains a copy (or symlink) to a specific interpreter. When you install packages into a virtual environment it will end up in this new folder, and thus isolated from other packages used by other workspaces".**
Per la definizione ufficiale di "virtual environment" vedi qui.
[NB. **In senso stretto, i *virtual environment* sono quelli creati con *venv* – ma anche quelli creati con Conda (formalmente definiti come *conda environments*) sono ambienti virtuali, e migliori! Conda inoltre è anche un gestore di package!]**

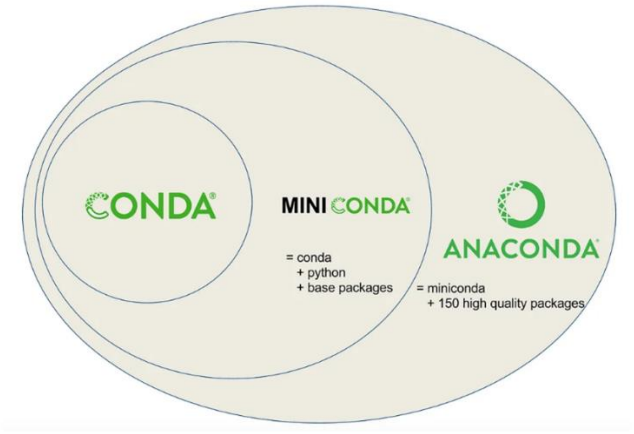
> *venv* è semplice e leggero, è **nativo** (è il modulo standard di Python per creare ambienti virtuali, fa parte di Python) e dunque non deve essere installato. Ma *conda* è potente e flessibile ed è sempre quello consigliato. Inoltre *venv* usa solo *pip* come gestore di package.

Il modo più semplice, lineare e affidabile per installare *conda* oggi è usare **Miniconda**.

Cosa è Miniconda?

Miniconda is a free, miniature installation of Anaconda Distribution that includes only conda, Python, the packages they both depend on, and a small number of other useful packages (da https://www.anaconda.com/docs/getting-started/miniconda/main)

La seguente figura illustra le **relazioni** tra conda, miniconda e anaconda:



Source: Planemo documentation

Per un **confronto** tra Anaconda, Miniconda e Conda vedi qui.

Cosa è incluso in Miniconda, in particolare quali package? Vedi qui.

Miniconda è quindi la soluzione più "classica": essenziale, pulita, senza software superfluo.

AI Overview

`venv` è integrato in Python e isola pacchetti Python da PyPI, ideale per sviluppo web e app; **Conda** è un gestore di ambienti e pacchetti più potente, non solo per Python (gestisce anche C++, R), con un repository vasto (Anaconda Cloud), ottimo per Data Science/ML, gestisce versioni di Python e dipendenze complesse, ma richiede l'installazione di Anaconda/Miniconda. La differenza principale è che `venv` è focalizzato su Python standard, mentre Conda è una piattaforma completa per la gestione di ambienti scientifici e linguaggi multipli.

Quando usare quale?

- **Usa `venv` se**: Lavori principalmente con Python, non hai bisogno di pacchetti specifici per la scienza dei dati e vuoi qualcosa di leggero integrato.
- **Usa Conda se**: Fai data science, machine learning, hai bisogno di gestire pacchetti complessi o versioni specifiche di Python/altre lingue, o hai bisogno di una gestione centralizzata di ambienti scientifici.

- Perché Miniconda?
    - installa solo conda + Python
    - scelta più semplice e pulita
    - un ambiente conda per progetto
    - niente installazioni nell'ambiente base
    - nessun pacchetto preinstallato inutile
    - massimo controllo sugli ambienti
    - meno problemi nel tempo (soprattutto su Windows)

- Anaconda è una distribuzione "completa" ma è più pesante e, per molti, oggi superflua. [4]

Ecco i **passi da seguire per installare miniconda:**

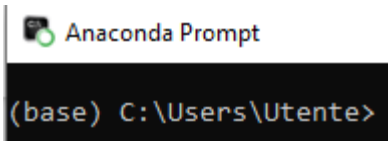
1. Installare Miniconda da qui
2. premere il bottone verde "Download" in alto a destra
3. premere il bottone verde "Get Started"
4. inserire le proprie credenziali (qualsiasi)
5. nella pagina di "Welcome to Anaconda" individuare il riquadro a destra per Miniconda
6. premere il link *Windows 64-bit Graphycal Installer*
7. eseguire il wizard di installazione:
    a. scegliere l'opzione *Just me*
    b. scegliere la directory di installazione (quella di default va bene)
    c. scegliere l'opzione avanzata *Add Miniconda3 to my PATH environment variable.* Anche se sconsigliata dal wizard, è l'opzione giusta per la **PRIMA** installazione di Python. In questo modo, potremo eseguire Python da QUALSIASI directory DOS, semplificandoci molto la vita.
    d. poi spuntare "Register Miniconda as default Python"
8. quindi, verificare la corretta installazione di miniconda in questo modo:
    - Dal menù Windows [5]:
        - attivare un "**Anaconda Prompt**" (non il prompt classico, attivabile digitando *cmd* nella barra di ricerca del menù Windows), che è reso disponibile dall'installazione di Miniconda. Si apre una finestra **terminale** di questo tipo:

[4] Vantaggi di Anaconda:
- ha Python
- ha Anaconda Navigator (la User Interface di ambiente)
- ha molte notebook interface e IDE
- ha molti package Python già installati
- permette di installare / deinstallare / aggiornare facilmente i package Python
- permette di passare da un ambiente virtuale Python ad un altro facilmente

Attenzione: per installare Anaconda occorre che il PC/VM abbia **almeno 16 GB di RAM**; vedi questa chat.

Se si installa Anaconda, nel wizard di installazione NON selezionare il checkbox dell'aggiornamento della variabile di ambiente di Windows PATH perché può creare fraintendimenti sugli eseguibili attivati.

[5] il tasto all'estrema sinistra in basso 



dove il nome *Utente* sarà differente per ogni utente.

- e sul terminale che appare, digitare: *conda --version*

- Se l'output a terminale mostra la versione **25.9.1** (a dicembre 2025) o superiori, *conda* è installato correttamente.

9. Verificare quali ambienti virtuali Python (VE) di conda sono presenti sul PC con questo comando:

   conda env list

Qui sotto una descrizione più dettagliata di **cosa è *conda***:

Search Labs | AI Overview

Conda is an open-source, cross-platform package and environment management system, originally developed by Anaconda but now maintained by the open-source community. It's used to install, run, update, and manage packages, along with their dependencies, across different languages and operating systems, especially popular in data science, machine learning, and AI.

Key features and benefits of Conda:

Language-agnostic:
While initially developed for Python, Conda can manage packages for various programming languages like R, Java, and C/C++.

Environment management:
Conda allows users to create and manage separate environments with specific versions of Python and other libraries, resolving dependency conflicts and ensuring reproducibility.

Package management:
Conda can install, update, and uninstall packages and their dependencies, ensuring consistent package installations and versions across different projects.

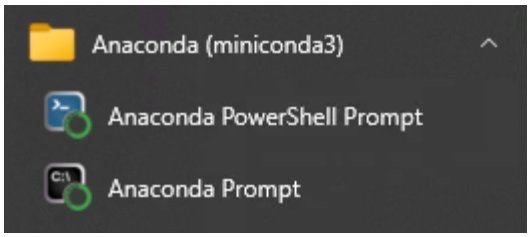
Cross-platform:
Conda works seamlessly on Windows, macOS, and Linux operating systems.

Unified interface:
Conda provides a unified interface for managing packages across different programming languages, simplifying workflows for developers and researchers.

In essence, Conda provides a comprehensive solution for managing software packages and their dependencies, particularly beneficial for complex projects involving multiple languages and libraries.

I PROMPT DI ANACONDA

L'installazione di Miniconda (o Anaconda) o rende disponibili sul PC/VM i prompt di Anaconda.



Cosa è un prompt Anaconda? E' un terminale DOS (quello attivabile con "cmd") con alcune caratteristiche in più. Vediamo la seguente risposta di chatGPT.

qual è la differenza tra un prompt anaconda e un terminale DOS?

Bella domanda! Vediamolo con un confronto semplice e diretto tra **Anaconda Prompt** e il classico **Prompt dei comandi di Windows (cmd.exe, detto anche terminale DOS)**:

⚙️ 1. Prompt dei comandi (cmd.exe)

- È il terminale **predefinito** di Windows.
- Fornisce accesso diretto alla shell DOS/Windows.
- Puoi eseguire comandi come `cd`, `dir`, `copy`, `ipconfig`, ecc.
- Non ha **Python**, **Conda** o **ambienti virtuali** attivati di default (a meno che tu non li configuri tu manualmente).

🐍 2. Anaconda Prompt

- È **una versione "custom" di cmd.exe** fornita da Anaconda.
- Quando lo apri:
  - Attiva automaticamente **l'ambiente Conda "base"**
  - Aggiunge al `PATH` tutte le **utility di Anaconda** (`conda`, `python`, `jupyter`, ecc.)
- È fatto apposta per lavorare in un contesto Pyth… data science.

Esempio:

Nel prompt DOS se scrivi `conda`, potresti ricevere errore ("comando non riconosciuto").

Nel **Anaconda Prompt**, invece, funziona subito.

🔄 Esempio pratico

Comando	Prompt DOS	Anaconda Prompt
`python`	Errore o Python di sistema	Python gestito da Conda
`conda list`	Errore	Mostra pacchetti attivi
`jupyter notebook`	Errore o non trovato	Parte il notebook
`pip install`	Usa pip globale	Usa pip nell'ambiente attivo

📦 Altre alternative

- **PowerShell** → Shell più moderna di Windows, simile a cmd ma con sintassi avanzata
- **Anaconda PowerShell Prompt** → come Anaconda Prompt, ma basato su PowerShell
- **Git Bash** o **WSL** → ambienti Unix-like su Windows, utili per chi lavora con tool Linux

🔍 In sintesi

Tool	Base	Ha Conda attivo	Scopo principale
Prompt DOS	cmd	❌	Comandi Windows generici
PowerShell	ps	❌	Shell avanzata Windows
Anaconda Prompt	cmd	✅	Python/Data Science con Conda
Anaconda PowerShell	ps	✅	Idem, ma su PowerShell

CREAZIONE DELL'AMBIENTE VIRTUALE DEDICATO

Dopo avere installato (correttamente?) Miniconda abbiamo dunque a disposizione anche conda.

Dunque ora **utilizziamo *conda*** per creare il nostro nuovo ambiente virtuale, che per genericità chiamiamo qui ***myenv*** (il "mio ambiente").

Il primo utilizzo corretto di *conda*, infatti, è proprio la creazione del primo VE (di tipo conda).

Vediamo i **passi operativi** per creare un nuovo VE python con *conda*:

Passo 1.

PRIMA di creare un nuovo ambiente virtuale ad hoc per il nostro progetto verifichiamo che esso NON esista già:

- ***conda env list*** → fornisce la lista degli ambienti virtuali (conda) già presenti sul nostro PC. Si dovrebbe ottenere **il seguente output** (a meno del nome utente contenuto nel prompt, che è specifico per ogni utente, qui genericamente *Utente*):

```
(base) C:\Users\Utente>conda env list

# conda environments:
#
# *  -> active
# + -> frozen
base                  *   C:\Users\Utente\miniconda3


(base) C:\Users\Utente>
```

  Che cos'è questo ambiente ***base***?

In `conda`, l'ambiente `base` è l'**ambiente "di sistema" di Conda**, cioè quello che viene creato automaticamente **quando si installa Miniconda/Anaconda** e che contiene:

- l'**eseguibile** `conda` stesso (e i suoi componenti: solver, librerie, ecc.)
- un **Python** (di solito sì: spesso `base` include Python, ma è un dettaglio "di comodo", non l'essenza)
- strumenti di gestione (pip può esserci o meno, dipende da come è stato installato)

A cosa serve davvero

- È l'**ambiente da cui Conda amministra tutto il resto** (creazione/attivazione di altri env, installazioni, aggiornamenti).
- È l'ambiente che Conda **attiva di default** quando si apre il terminale *se* è abilitata l'auto-attivazione.

Perché si vede in `conda env list`

Perché è un ambiente a tutti gli effetti, solo che è **speciale**: è quello "radice" collegato alla directory dell'installazione (es. `...\miniconda3` o `...\anaconda3`).

Buona pratica (molto consigliata)

- Usare `base` **solo per Conda** (aggiornare conda, creare env).
- Installare librerie e progetti **in ambienti separati** (`conda create -n progetto ...`), così si evita di "sporcare" l'ambiente fondamentale e ridurre conflitti.

Nota pratica: auto-attivazione di `base`

Se non si vuole che `base` si attivi automaticamente all'apertura del terminale:

bash Copia codice

```
conda config --set auto_activate_base false
```

Il comando *conda env list*, se su quel PC non si è mai lavorato con Python e/o con *conda*, dovrebbe **elencare solo l'ambiente *base***.

Solo uno degli ambienti elencati può essere ATTIVO. Lo si riconosce facilmente dalla presenza di un * nella lista, come visto prima, e dal **prompt del terminale (che inizia con il suo nome in parentesi)**. Ad esempio:

```
(base) C:\Users\Utente>
```

(il nome utente nel prompt è ovviamente differente per ogni utente)

- Se fosse già presente nella lista anche l'ambiente ***myenv***, che vogliamo creare, se esso è vecchio o ha problemi, per sicurezza **rimuoviamolo** in questo modo:
  a. come detto prima, un'operazione di gestione ambienti di questo tipo DEVE essere eseguita dall'ambiente *base*
  b. **se l'ambiente attivo è *base***, bene possiamo passare al passo c. Se l'ambiente attivo NON è *base*, **deattiviamo** l'ambiente (corrente) con ***conda deactivate***, oppure attiviamo l'ambiente *base* con *conda activate base*
  c. **dall'ambiente *base* rimuoviamo** l'ambiente ***myenv*** con: ***conda remove --name myenv --all* (come amministratore)**
  d. *conda env list* fornisce ora solo l'ambiente *base*.

NB. A prescindere dall'esito della rimozione dell'ambiente (terminato correttamente oppure no), è bene verificare che la cartella del VE non sia ancora presente (può succedere!) ed in questo caso **cancellarla** fisicamente (sempre dal prompt anaconda), in questo modo:

rmdir /s nome_cartella

La cartella fisica dell'EV è in genere:

C:\Users\<nome_utente>\miniconda3\envs\<nome_ambiente> **- in Miniconda**

C:\Users\<nome_utente>\anaconda3\envs\<nome_ambiente> **- in Anaconda**

Si può trovare la cartella RADICE (*C:\Users\<nome_utente>\miniconda3*) in questo semplice modo: ***conda info --envs.*** E poi, una volta posizionati in essa, fare *cd envs*.

⚠ Sempre come pulizia iniziale, è bene cancellare **altri** ambienti virtuali che fossero presenti nella lista (a parte quello *base*) e non fossero nostri oppure fossero stati creati in tempi molto precedenti. La presenza di ambienti inutili o vecchi è infatti **un elemento di inutile complicazione**.

Per sicurezza, se un VE può essere di nostro interesse in futuro, lo possiamo esportare prima di cancellarlo: *conda env export -n NOMEENV > NOMEENV.yml.*

E poi, in seguito, eventualmente ricrearlo con: *conda env create -f NOMEENV.yml*

Passo 2

- Siamo ora pronti a creare l'ambiente virtuale dedicato ***myenv*** (il nome è a vostra scelta), installando in esso una versione "stabile" di Python (ad esempio la 3.11, oppure quella richiesta dall'applicazione, non una delle ultime!), in questo modo:

  conda create -n myenv python=3.11

  Sono richieste alcune conferme. Il download dei vari moduli richiede qualche decina di secondi.

  Se l'operazione è andata a buon fine la parte finale dell'output sul terminale DEVE contenere le seguenti informazioni con i 3 *done* (a parte i suggerimenti finali, commentati):

```
Downloading and Extracting Packages:

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

Point Se non ci sono vincoli applicativi, si consiglia di installare nell'ambiente virtuale la versione **python 3.11** per progetti nuovi, la versione **3.10** per progetti tradizionali. Infatti:

- **Python 3.11** → miglior equilibrio: moderno, veloce, compatibilità ampia.
- **Python 3.10** → la scelta "più conservativa": massima compatibilità con librerie e tool un po' datati (utile se si vogliono zero sorprese).

- **Python 3.12** → va benissimo solo se si è certi che tutte le librerie del progetto (in particolare quelle di ML/AI, in primis *PyTorch* e *TensorFlow*) siano pienamente compatibili; altrimenti è quella che più spesso crea attriti.
- **Evitare per il momento Python 3.13 e Python 3.14 (troppo recenti)**
- **Regola generale**: se si vuole usare **PyTorch** o **TensorFlow** (per le reti neurali) NON installare l'ultima versione di Python, a maggio 2025 la 3.13.3, perchè Pytorch e TensorFlow non sono subito "portati" su di essa (servono in genere alcuni mesi od anche più). Meglio installare una versione di Python meno recente, ad esempio la 3.11. Per chi scrive software gestionale e scientifico non si perde nulla [6].

Passo 3

Verificare la corretta creazione dell'ambiente *myenv*, digitando dal terminale il solito comando *conda env list*. Si deve ottenere la seguente lista:

```
# conda environments:
#
# *  -> active
# + -> frozen
base                  *   C:\Users\Utente\miniconda3
myenv                     C:\Users\Utente\miniconda3\envs\myenv
```

che riporta la presenza, sotto la cartella *envs*, della sottocartella dedicata all'ambiente *myenv* appena creato.

Passo 4

Ora, si deve attivare il nuovo VE, digitando sul terminale:

conda activate ***myenv***

Attenzione: ora il prompt diventa "*myenv*", ad indicare appunto che ora è questo l'ambiente attivo, e non più *base*!

```
(base) C:\Users\Utente>conda activate myenv

(myenv) C:\Users\Utente>
```

Passo 5

Verificare che la versione Python installata nell'ambiente sia davvero 3.11 e che punti a *myenv*

python –version

[6] Il libro di Raschka (di cui al video di sopra) comunque è stato testato sulle versioni 3.10, 3.11, 3.12 e 3.13]. La 3.11.11 è di dicembre 2024.

```
(myenv) C:\Users\Utente>python --version
Python 3.11.14
```

ok!

python -c "import sys; print(sys.executable)"

```
(myenv) C:\Users\Utente>python -c "import sys; print(sys.executable)"
C:\Users\Utente\miniconda3\envs\myenv\python.exe
```

sys.executable deve finire in ***...\miniconda3\envs\myenv\python.exe***

Qual è il significato di questi due comandi?

Il primo comando (*python --version*) fa una sola cosa: **stampa la versione del Python che viene avviato dal comando python in quel momento**.

Esempi tipici di output:

- Python 3.11.9
- Python 3.13.2

È utile perché **conferma subito se si sta usando python 3.11** (ad esempio dentro l'ambiente *myenv*) oppure un'altra versione (*base* di Miniconda, Python standalone, ecc.).

Il secondo comando (*python -c "import sys; print(sys.executable)")* serve a far stampare a Python **il percorso completo dell'eseguibile** che è stato avviato in quel momento.

- -c "..." significa: "esegui questo piccolo pezzo di codice Python direttamente da riga di comando, senza aprire un file".
- sys.executable è il **path del python.exe realmente usato**.

Per esempio, dopo aver fatto prima *conda activate myenv* ci si aspetta da questo comando qualcosa tipo:

C:\Users\Utente\miniconda3\envs\myenv\python.exe

Se invece il comando restituisce ...\miniconda3\python.exe (base) o ...\Python311\python.exe (installazione standalone), allora **non si sta usando l'ambiente che si credeva di avere attivo**.

La differenza tra i due comandi:

- *python --version* dice **quale versione di Python si sta usando**.
- *python -c "import sys; print(sys.executable)"* dice **quale eseguibile/percorso** (quindi quale installazione/ambiente).

Passo 6

Conda non è solo un gestore di ambienti virtuali, ma **è anche un gestore di package Python** (per installazione, disinstallazione, upgrade dei vari package). Un'alternativa come gestore di package (ma non di ambienti!), con pro e contro rispetto a *conda*, è il modulo *pip*, "the **p**ackage **i**nstaller for **p**ython). Conviene installare nell'ambiente *myenv* anche *pip*.

Ecco il consiglio in merito di chatGPT 5.2:

1) È utile installare anche pip?

Quasi sempre **sì**, perché:

- alcuni pacchetti stanno solo su **PyPI** → quindi serve `pip`;
- anche se usi soprattutto `conda`, `pip` è una "ruota di scorta" utile quando un pacchetto non è disponibile (o non nella versione giusta) sui canali conda.

Se sai già che installerai *solo* roba presente su `conda-forge` / `defaults`, allora potresti anche farne a meno… ma nella pratica prima o poi torna comodo.

2) Lo installo con conda o con pip (pip install pip)?

Installalo con `conda`.
`pip install pip` di solito non ha senso e può creare confusione (pip aggiornato "a modo suo" dentro un ambiente gestito da conda).

Comandi consigliati:

bash Copia codice

```
conda install -n myenv pip
```

Oppure, se hai già attivato `myenv`:

bash Copia codice

```
conda install pip
```

Regola pratica per installare pacchetti dopo

- **Prima prova con** `conda install ...` (meglio se ti basi su un canale coerente, tipo `conda-forge`).
- **Se non c'è o non va bene la versione**, allora `pip install ...`.
- Quando usi `pip`, fallo **con l'ambiente attivo** e possibilmente **dopo** aver installato conda tutto ciò che può.

Eseguiamo dunque dal solito terminale (**sotto *myenv***):

conda install pip

E' possibile che, al termine della installazione del *pip*, *conda* vi suggerisca di aggiornare la sua versione [7]. In questo caso accettiamo l'invito ed eseguiamo (da qualsiasi ambiente attivo)

conda activate base (perché la gestione di *conda* e degli ambienti deve sempre essere fatta da *base*)

conda update -n base -c defaults conda

[7] Questo avviso è **mostrato quando *conda* fa una qualunque operazione di contatto con i canali di distribuzione dei package python** (install, update, create, ecc.), perché in quel momento:

1. *conda* **interroga i canali** (defaults, eventuali altri) per scaricare i metadati dei pacchetti
2. mentre lo fa, **vede anche la versione più recente di *conda*** disponibile sui canali configurati
3. se la versione installata (magari giorni o settimane prima) è più vecchia, stampa l'avviso

In sintesi: l'avviso è un "promemoria" che compare **solo quando conda ha occasione di verificare online** (come nel caso di *conda install pip*)

Come sempre, l'output del terminale, se corretto, deve riportare alla fine:

```
Downloading and Extracting Packages:

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

JUPYTER NOTEBOOK

Se **si vuole aprirein VSC un notebook Jupyter** (è un formato di file testuale JSON, di suffisso *.ipynb*, molto utilizzato nel mondo python) oppure se si desidera **installare l'IDE Jupyter Notebook**, un'alternativa a VSC (non consigliata ma possibile per chi sia abituato a questa IDE), la prassi classica (dall'ambiente ***myenv* attivo**) è installare **l'estensione VSC di nome "Jupyter" di Microsoft.**

In genere VS Code propone automaticamente queste estensioni quando si apre un file *.ipynb*, ma è meglio installarli esplicitamente per poterli selezionare.

Se si desidera anche installare l'IDE Jupyter Notebook - in questa guida consigliamo di utilizzare VSC, come detto diverse volte - allora occorre procedere nel seguente modo (sempre dall'ambiente *myenv* attivo):

a) installare la IDE Jupyter Notebook, in questo modo:

conda install -c conda-forge notebook ipykernel (se si usa ***conda***, come consigliato)

dove *conda-forge* [8] è il principale canale di distribuzione dei package python

oppure:

pip install notebook ipykernel (se si usa ***pip***)

[8] **Conda-forge** è un'organizzazione e un progetto comunitario open-source su GitHub che fornisce un'ampia collezione di ricette (ricette) per creare e distribuire pacchetti software per l'ecosistema Conda, ampliando enormemente la disponibilità di librerie scientifiche, di data science, e di vari altri strumenti, spesso più aggiornati o specifici rispetto al canale predefinito di Anaconda. Funziona come un canale aggiuntivo (channel) per il gestore di pacchetti e ambienti Conda, offrendo un modo per installare software non inclusi nei repository standard, gestendo dipendenze complesse in modo multipiattaforma.

In sintesi:

- **Cosa è:** Un'infrastruttura comunitaria guidata da volontari per creare, mantenere e distribuire pacchetti Conda.
- **Scopo:** Fornire pacchetti software (inclusi Python, R, e altro) che potrebbero non essere nel canale predefinito di Anaconda, con aggiornamenti frequenti e supporto per diverse piattaforme (Windows, macOS, Linux).
- **Come funziona:** Contiene "feedstocks" (ricette) che vengono utilizzate per costruire pacchetti (built distributions) e caricarli su Anaconda.org.
- **Utilizzo:** Gli utenti aggiungono conda-forge come canale (channel) al proprio ambiente Conda per accedere a questi pacchetti aggiuntivi, spesso anteponendolo al canale di default per priorità.
- **Vantaggi:** Accesso a un catalogo molto più vasto e aggiornato di pacchetti, ideale per data science, machine learning e calcolo scientifico, garantendo al contempo isolamento degli ambienti per evitare conflitti.

È gestito come progetto sponsorizzato da NumFOCUS e si integra perfettamente con Conda, il gestore di pacchetti open-source multi-linguaggio.

a. registrare il kernel (è utile quando Jupyter Notebook o VSC non lo "vedono" subito)

python -m ipykernel install --user --name myenv --display-name "Python (myenv)"

Da lì Jupyter Notebook (e anche VSC) vedrà un kernel di nome "Python (myenv)".

INTEGRAZIONE CON VS CODE

A questo punto, e solo a questo punto (cioè sempre dopo aver installato Miniconda [9] più le altre cose):

- **aprire VS Code**
- *View* (dal menù in alto) → *Command Palette*
- *Python: Select Interpreter* (o *Select kernel* se si è installato un kernel jupyter)
- scegliere …*envs**mioenv**python.exe* → VSC userà quell'interprete

INSTALLARE I PACKAGE NECESSARI NELL'AMBIENTE VIRTUALE DEDICATO

Da questo momento:

- Installare nell'ambiente virtuale dedicato (*myenv*) i **package necessari** (con la versione richiesta).
- Utilizzare preferenzialmente *conda* (che, come detto prima, oltre ad essere un gestore di VE è anche un gestore di package), con questa sintassi:

---

[9] Occorre fare così perché VS Code (in particolare l'estensione Python**) "fotografa" gli interpreti disponibili** e, se conda non è ancora installato, non può rilevare:

- il comando conda
- la cartella degli ambienti (…\miniconda3\envs\…)
- gli interpreti Python dentro quegli ambienti

Se VS Code era già aperto mentre installava conda, spesso succede che:

- la lista in Python: Select Interpreter non mostra gli ambienti conda
- il terminale integrato non "vede" subito conda
- le auto-attivazioni dell'ambiente non partono

Chiudere e riaprire VS Code (o almeno riavviare la finestra: Ctrl+Shift+P → Developer: Reload Window) forza VS Code a:

- rileggere PATH/registrazioni di sistema
- riscoprire gli interpreti Python
- aggiornare la lista degli ambienti disponibili

In breve**: è un modo semplice per evitare che VS Code lavori "con la memoria vecchia" dell'installazione precedente.**

conda install -c conda-forge <nome-package>

In VS Code può succedere che il terminale dica "(myenv)" ma il comando *pip* punti a un altro Python (base o un'altra installazione). Si verifica così:

python -m pip -V

python -m pip freeze

📌 **Consiglio:** salva i pacchetti in *requirements.txt*:

pip freeze > requirements.txt

La guida termina qui. I capitoli successivi forniscono chiarimenti tecnici su alcuni strumenti incontrati prima.

COSA È L'INTERPRETE PYTHON?

Interprete Python: è l'eseguibile *python.exe* (o *python*) che esegue **script .py**, debug, terminale, ecc. In VS Code è ciò che si seleziona con *Python: Select Interpreter*, come visto prima.

In VS Code "l'interprete" è l'eseguibile Python che verrà usato quando si preme Run, quando si avvia il debug, quando il terminale lancia python, e quando le estensioni analizzano/importano i pacchetti.

L'interprete "giusto" è semplicemente quello che:

- corrisponde all'**ambiente** in cui vuole lavorare (progetto specifico o sistema)
- ha i **pacchetti** necessari già installati (numpy, pandas, ecc.)
- ha una **versione Python** compatibile con il suo codice/librerie

Esempi tipici (in Windows):

- *C:\Users\<utente>\AppData\Local\Programs\Python\Python312\python.exe*

→ Python "di sistema" installato da python.org (spesso va bene per progetti semplici).

- C:\Users\<utente>\anaconda3\python.exe

→ base Anaconda (da usare con cautela: meglio ambienti dedicati).

- C:\Users\<utente>\anaconda3\envs\mioenv\python.exe

→ ambiente Conda dedicato (di solito la scelta migliore per stabilità dei progetti).

- <cartella_progetto>\.venv\Scripts\python.exe

→ virtual environment del progetto (molto "pulito" e tradizionale).

Come detto - lo ribadiamo - si sceglie l'interprete in VS Code in questo modo:

1. Ctrl+Shift+P

2. **Python: Select Interpreter**

3. Selezionare quello che punta all'ambiente desiderato (spesso riconoscibile dal nome dell'env e dal path).

Un controllo rapido per capire se è quello giusto:

- aprire il terminale integrato di VS Code [10] e lanciare:

  python -c "import sys; print(sys.executable)"

  deve stampare un percorso tipo ...\miniconda3\envs\mioenv\python.exe

- verificare

  python -m pip list (per vedere i pacchetti disponibili)

Se si usa conda, "l'interprete giusto" è quasi sempre il Python dentro l'ambiente conda del progetto, non quello "base".

Evitare, quando possibile, …\miniconda3\python.exe (ambiente base) per lavorare: tende a diventare un "cassetto" dove finisce di tutto e col tempo crea conflitti.

Se non compare, spesso basta:

- aprire VS Code dopo aver creato l'ambiente, e
- assicurarsi che l'estensione Python sia installata.

Quindi:

conda info --envs

deve risultare attivo (*) lo stesso ambiente che sta usando VS Code

➔ NB. Nel caso di creazione di un nuovo ambiente virtuale python e dell'eventuale installazione in esso di una differente versione di python, il comportamento dei due comandi DOS 'where

[10] Per cambiare la cartella in cui si apre il terminale integrato di VS Code, la via "classica" è impostare **Terminal › Integrated: Cwd**.
Metodo da interfaccia (consigliato)

1. **File → Preferences → Settings** (su Windows: *Ctrl+,*)
2. Nella barra di ricerca scrivere: **Terminal Integrated Cwd**
3. Trovare **Terminal › Integrated: Cwd**
4. Inserireil percorso desiderato, ad es. C:\Users\Utente\Desktop\progetti

python' e 'python --version' necessita di essere chiarito**. Vedi la sezione ad hoc: "Versioni multiple di Python".**

I GESTORI DEI PACKAGE

I package ampliano il funzionamento di python base. L'indice ufficiale è qui.

Un package manager installa, disinstalla e aggiorna i package di Python.

Ce ne sono diversi:

- **pip**: quello tradizionale, attinge sempre all'indice ufficiale dei package Pyton PyPI e perciò trova sempre i package!
- **conda** (è anche un gestore di ambienti virtuali): più lento di pip ma gestisce meglio i conflitti tra i package. Purtroppo a volte non trova i package (soprattutto quelli più piccoli o più recenti) perché **NON accede a PyPI** ma a canali di distribuzione suoi (il migliore è *conda-forge*). Li si può impostare (per la singola install) con il parametro *-c* (ad esempio: *conda install xxxxx -c conda-forge*). Molti package non disponibili nel canale *defaults* lo sono invece nel canale *conda-forge*.
- **mamba**: molto veloce
- **uv:** il migliore, vedi capitolo a fine documento.

Regola generale (consigliata da molti):

- **Per prima cosa provare sempre a installare il package da *conda***, soprattutto per i package scientifici, di sistema, per big data ML e DL. Se il package è disponibile nei canali di *conda*, *conda* sa gestire le dipendenze tra package molto meglio di *pip*.
- Se il package non è disponibile su *conda*, installarlo con *pip*
- Attenzione: mescolare installazioni con *conda* e *pip* nello stesso ambiente virtuale può portare a conflitti ed è dunque sconsigliabile. Tuttavia, ciò NON è un motivo per installare tutto da *pip*!! **Provare sempre prima con *conda* e poi con *pip***, i benefici sono superiori ai problemi eventuali (mix di installazioni differenti).

CREAZIONE DEL PRIMO FILE

Creare un file .py con print hello world e qualche import, fare run.

Per eseguire un file .py in Visual Studio Code ci sono tre modi "classici".

1) Pulsante PLAY ("Run Python File")

- Aprire il file .py
- In alto a destra cliccare "Run Python File"
- VS Code esegue lo script nel terminal integrato

Se il file non parte come previsto, quasi sempre è perché è selezionato l'interprete sbagliato: *Ctrl+Shift+P* → **Python: Select Interpreter** e scegliere l'ambiente conda giusto.

2) Dal terminale integrato (metodo più sicuro)

- Terminal → New Terminal
- Attivare l'ambiente conda (se non è già attivo):

  conda activate NOME_ENV

- Posizionarsi nella cartella del file (se serve):

  cd percorso\cartella

- Eseguire:

  python nomefile.py

Vantaggio: si vede esattamente quale python sta usando.

3) "Run/Debug" (utile se si vuole il debugger)

- Aprire il file .py
- Premere F5 (Run and Debug)
- Se richiesto, scegliere Python File

Poi si possono usare breakpoint, step, variabili, ecc.

APRIRE ED ESEGUIRE JUPYTER NOTEBOOK

Per aprire ed eseguire un notebook Jupyter in Visual Studio Code servono, in pratica, **tre cose**: estensioni giuste, un Python "vero" installato, e un kernel Jupyter [11] nel relativo ambiente.

1) Estensioni in VS Code

- **Python** (Microsoft)
- **Jupyter** (Microsoft)

Senza queste, il file .ipynb si apre ma non riesce a eseguire le celle correttamente.

[11] Kernel (Jupyter): è il "motore" che esegue le celle di un notebook .ipynb. Un kernel è legato a un ambiente/interprete, ma è gestito tramite Jupyter (ed è selezionabile dal notebook, in alto a destra, come "Kernel").

2) Un interprete Python installato (meglio se in un ambiente)

Va bene:

- **Python** "classico" (python.org) + venv
- **Anaconda/Miniconda** (molto comodo per notebook)

In VS Code poi si seleziona l'interprete: **Ctrl+Shift+P → "Python: Select Interpreter"**.

3) Il kernel Jupyter nell'ambiente selezionato

Nell'ambiente Python scelto deve essere presente **ipykernel** (e spesso anche jupyter).

Dobbiamo quindi **installare il kernel** (dal prompt anaconda, dall'ambiente virtuale *myenv* attivato) in due passi:

1. **ipykernel** è un "modulo" che collega python a Jupyter Notebook, consentendo a quest'ultimo di parlare con python. ipykernel è un kernel interattivo.

   Installare il kernel con: ***conda install ipykernel.*** L'output corretto a terminale del comando è:

   Installed kernelspec tf_cpu in C:\Users\Utente\AppData\Roaming\jupyter\kernels\tf_cpu

   NB. Si può ottenere la lista dei kernel jupyter installati con il comando da prompt: ***jupyter kernelspec list.***

   Per rimuovere un kernel jupyter installato si fa: ***jupyter kernelspec remove nome-kernel***

2. Il secondo comando, successivo, **<u>associa un kernel jupyter ad un VE</u>**, rendendo così l'ambiente virtuale disponibile come kernel in Jupyter (selezionabile in Jupyter Notebook). Si chiama anche "registrazione".

   python -m ipykernel install --user --name accenture --display-name "accenture"

   Ogni kernel jupyter associato ad un VE è una ISTANZA di ipykernel. Come per pip, ogni VE ha <u>così</u> la sua copia di ipykernel. A differenza delpip, tuttavia, ipykernel non è attivabile autonomamente, ma solo come modulo.

In un file .py NON esiste il concetto di "kernel" (quello è tipico dei notebook .ipynb). Quindi:

- **quando si esegue un .py, conta l'interprete Python selezionato (quello di *Python: Select Interpreter,* e in pratica anche quello mostrato nella status bar in basso);**

Flusso "tradizionale" di utilizzo

1. Aprire **la cartella di lavoro** (workspace) in VS Code.
2. Aprire il file .ipynb.
3. In alto a destra scegliere il **Kernel** (deve corrispondere all'ambiente Python).
4. Eseguire una cella (Run).

Se, dopo aver fatto questo, VS Code continua a non trovare kernel o dà errori, di solito è perché **l'ambiente selezionato non è quello in cui è stato installato ipykernel**. In quel caso basta reinstallarlo nell'ambiente corretto e riselezionare il kernel.

INTERPRETE VS KERNEL (IN VSC)

1) Python: Select Interpreter (Command Palette)

Seleziona **l'interprete Python dell'ambiente di lavoro** (workspace [12]). In pratica influenza:

- esecuzione di **file .py** ("Run Python File", Debug, test)
- **terminali** creati da VS Code (quale python viene usato di default)
- **linting / type checking / IntelliSense** (Pylance)
- installazioni con pip/conda fatte *dal terminale integrato* (se quel terminale è attivato su quell'ambiente)

È una scelta "generale" per il progetto.

2) Bottone in alto a destra nel notebook → scelta Kernel

Cambia il **kernel Jupyter** che esegue *quel notebook* (.ipynb). Influenza:

- quale Python esegue le **celle**
- quali librerie sono disponibili *durante l'esecuzione del notebook*
- spesso viene salvato come preferenza del notebook (metadata) o della sessione

Può mostrare opzioni diverse perché un kernel può essere:

[12] In VS Code, **il "workspace"** è, in pratica, **la cartella di lavoro del progetto** (o un insieme di cartelle) che VS Code sta gestendo come unità.

- Se si apre una cartella con **File → Open Folder…**, quella cartella è il *workspace*.
- Se si apre un file "sparso" senza cartella, **non c'è un workspace**: VS Code lavora "senza progetto", e molte impostazioni (tipo l'interprete) diventano più ambigue.
- Esistono anche i **multi-root workspace**: un unico workspace con più cartelle, salvato in un file .code-workspace.

Per capirlo al volo:

- Guardare nella barra in alto nella finestra: il nome mostrato lì di solito è il *workspace* (cioè la cartella/progetto aperto).
- Oppure in Explorer a sinistra: se vede una cartella "radice" con nome, quello è il workspace.

- un ambiente locale con ipykernel
- un kernel registrato (kernelspec)
- talvolta anche kernel remoti / WSL / container (a seconda della configurazione)

La differenza pratica più importante

Si può avere, per esempio:

- Interpreter del progetto = spegea
- Kernel del notebook = base

In quel caso VS Code "vede" e suggerisce come se fosse spegea, **ma il notebook esegue in base** (e viceversa).

Come verificare senza dubbi

Nel notebook esegua una cella:

- *import sys*
- *sys.executable*

Quello è **il Python reale** usato dal kernel.

Se si vuole la regola "classica" che evita problemi: **Interpreter = ambiente del progetto**, **Kernel = stesso ambiente del progetto** (installando ipykernel nell'ambiente, se necessario).

INSTALLAZIONE IN VSC DELLE ESTENSIONI OPZIONALI MA UTILI.

Opzionali ma utili:

- Ruff (linting/formatting veloce; alternativa moderna a flake8/isort ecc.)
- Black Formatter (se preferisce formattazione "classica" e standardizzata)

Black Formatter serve a formattare automaticamente il codice Python in modo uniforme e coerente, senza dover perdere tempo a decidere "come" impaginare (spazi, a capo, indentazioni, lunghezza righe, ecc.).

In pratica:

- prende il file .py
- lo riscrive seguendo regole standard (molto diffuse nella comunità Python)
- così tutto il progetto ha lo stesso stile, anche se ci lavorano più persone

Cosa risolve concretamente:

- “guerre di stile” (virgolette singole vs doppie, dove andare a capo, ecc.)
- diff su Git pieni di cambi di spaziatura
- codice disomogeneo tra file e autori

In VS Code, con Black:

- si può formattare al salvataggio (format on save)
- oppure manualmente con Format Document

Cosa non fa:

- non trova bug logici
- non “migliora” l’algoritmo
- non è un linter (quello segnala errori/stile; Black formatta e basta)

Se si usa già Ruff, spesso oggi si usa la coppia:

- Black per formattare
- Ruff per linting (e, se configurato, anche alcune correzioni automatiche)

Vuoi che ti prepari **un file .md già formattato con questa guida e i link attivi**, così puoi convertirlo in PDF con un comando (pandoc) o direttamente da VS Code?

LA PROCEDURA DI CREAZIONE DI UN AMBIENTE VIRTUALE PYTHON CON IL PACKAGE UV

Un’alternativa a *pip* e *conda* come gestore di package, molto usata e di moda in questi ultimi due anni, è ***uv***.

0. Installazione di *uv*: **pip install uv** (sì, si può installare un nuovo package manager con uno vecchio!)

   uv venv --python=python3.10

   crea l'ambiente virtuale di nome .venv, che appunto è sempre cancellabile e ricreabile, qualsiasi cosa succeda.

Nella sotto-directory Script (anzichè bin, come nel video di Raschka) c'è l'eseguibile python. Se lo si attiva si vede che è la v. 3.10! [non serve fare prima *source activate*, come nel video].

Se si cancella la directory .venv (dall'explorer di Windows - è molto più semplice) e si esegue python --> è attivata l'altra versione.

1. Installazione dei package

   uv pip install packages

   uv pip install torch (il nome di PyTorch in Python è torch!).

   Ad ogni inizio notebook di capitolo sono installati i packages necessari.

   Nel file *requirements.txt* sono riportate le versioni minime.

```
torch >= 2.3.0              # all
jupyterlab >= 4.0           # all
tiktoken >= 0.5.1           # ch02; ch04; ch05
matplotlib >= 3.7.1         # ch04; ch06; ch07
tensorflow >= 2.18.0        # ch05; ch06; ch07
tqdm >= 4.66.1              # ch05; ch07
numpy >= 1.26, < 2.1        # dependency of several other
libraries like
                            # torch and pandas
pandas >= 2.2.1             # ch06
psutil >= 5.9.5             # ch07; already installed
automatically as
                            # dependency of torch
```

   Un modo molto facile per adattare tutte le versioni corrette è il seguente:

   - scaricare il file *requirements.txt* da github al file system di Windows

   - dalla directory di download (se non la si è aggiunta al path): *uv pip install -r requirements.txt –system*

3. Apertura IDE

   uv run jupyter lab

In Google Colab (un nuovo notebook):

!pip install uv

!uv pip install - r <raw-file>

dove <raw-file> è la url github della versione raw del file 'requirements.txt'

NB. ogni volta che apriamo un notebook google colab in una nuova sessione, è un nuovo ambiente virtuale.

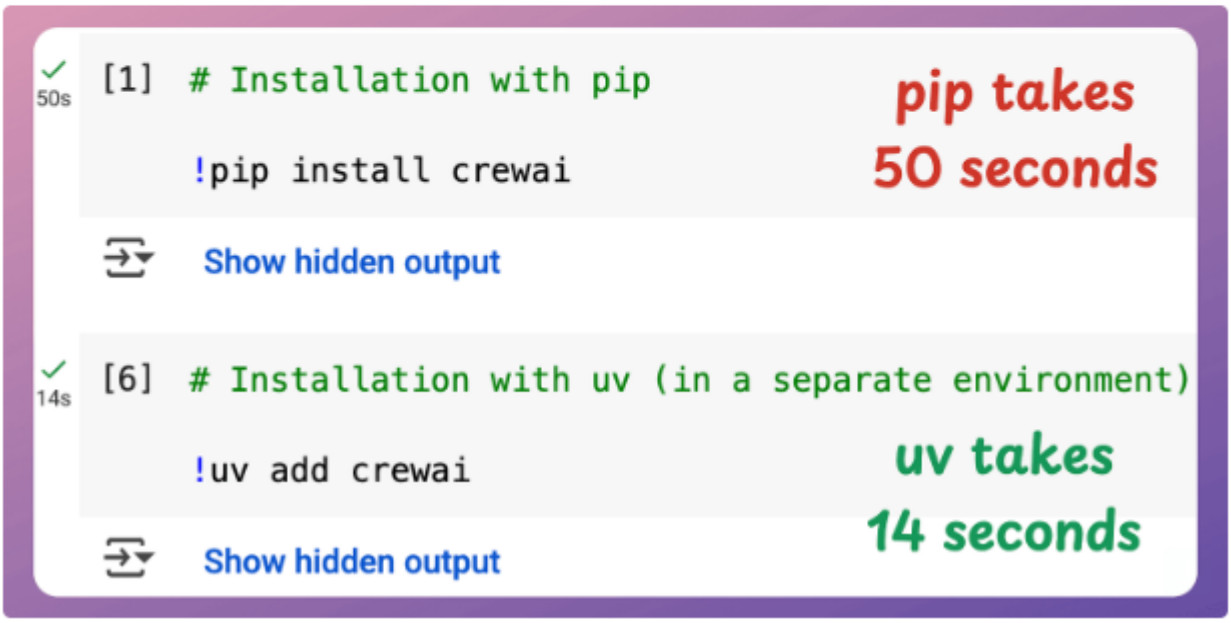
Cheesheet su UV (newsletters DDoDS – 13 settembre 2025)

PYTHON

uv cheatsheet and hands-on guide for Python devs

uv is incredibly fast.

- Creating virtual envs. using uv is ~80x faster than `python -m venv`.
- Package installation is 4–12x faster without caching, and ~100x with caching:



FONTI

Fonti (disponibili i sottotitoli in italiano):

- Install Miniconda (Python) with Jupyter Notebook and Setting Up Virtual Environments on Windows 10
- Set up your code environment (Raschka) – vedi anche github di RASBT → folder setup