

SQLServer da Python

(c) 2025 Antonio Piemontese

Aggiornamenti:

- Ad ottobre 2025 è disponibile la [preview](#) di **SQLServer 2025**.
- A ottobre 2025 lo standard SQL più recente è [quello del 2023](#)

La rilevazione dell'ambiente

Il presente notebook funziona sia in Jupyter Notebook che Google Colab. A questo scopo, per garantire la compatibilità di alcune celle di codice con entrambi gli ambienti indicati, **rileviamo l'ambiente di esecuzione del notebook** (Jupyter oppure Google Colab): la seguente cella imposta una **variabile booleana** `IN_COLAB` che sarà poi **testata** da alcune celle del notebook (in particolare quelle di visualizzazione di immagini e PDF) per personalizzarne il comportamento a seconda dell'ambiente di esecuzione dell'ambiente rilevato.

```
In [1]: # impostazione del TOGGLE BINARIO:
try:
    import google.colab                # package disponibile SOLO in Google Colab
    IN_COLAB = True
except ImportError:
    IN_COLAB = False

print("Siamo in Colab:", IN_COLAB)
```

Siamo in Colab: False

Import dei package per la visualizzazione delle immagini

```
In [3]: # package necessari per la VISUALIZZAZIONE delle IMMAGINI e dei PDF (necessari s
# IPython.display è la cassetta degli attrezzi di Jupyter/Colab per mostrare con
# direttamente nell'output della cella. Per maggiori info al riguardo vedi il no
from IPython.display import Image, display, IFrame

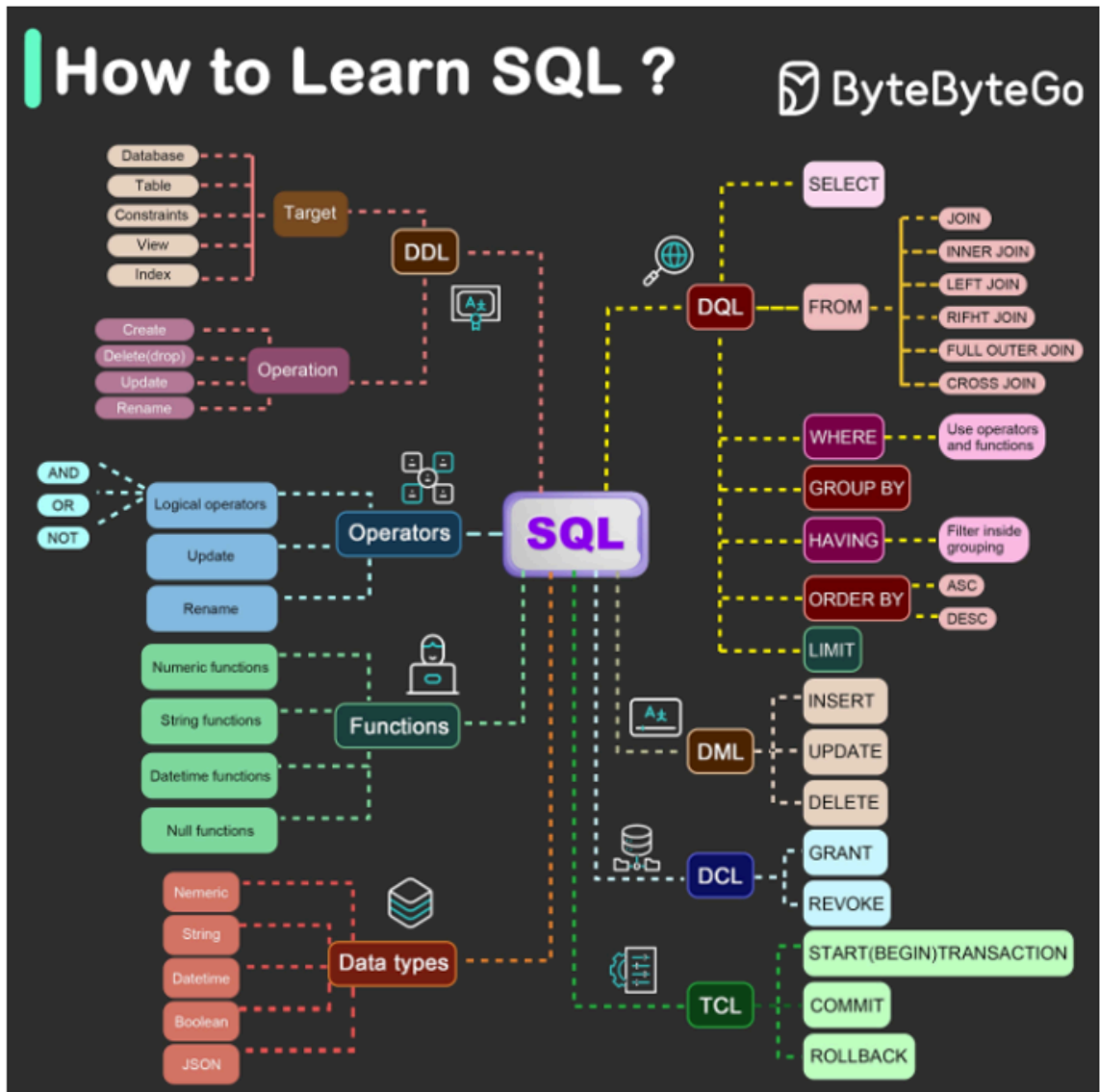
import os                                # necessario a Google Colab
                                         # i contenuti del 'content'

if IN_COLAB:
    from google.colab import files
```

SQL: introduzione

Vedi la [voce Wikipedia](#).

```
In [4]: Image('SQL_sinossi.png') if IN_COLAB else display(Image(filename='SQL_sinossi.png'))
```



Le librerie (package) python

Per lavorare con SQLServer da Python abbiamo bisogno di una libreria per stabilire la connessione e lavorare con il motore del DB.

Abbiamo alcune alternative, due in particolare: `pyodbc` e `sqlalchemy`

pyodbc

Cos'è `pyodbc`? Vedi [qui](#) nel *pypi*. `pyodbc` implementa la [Python Database API Specification v2.0](#), come anche fa SQLite3 (vedi [questa chat](#)).

`pyodbc` è una libreria Python che permette di connettersi a database relazionali tramite

il [driver ODBC \(Open Database Connectivity\)](#).

Non confondere ODBC (il driver) con Python Database API Specification v2.0. (l'interfaccia applicativa).

Cosa permette di fare:

- aprire una connessione verso SQL Server, MySQL, PostgreSQL, SQLite, ecc.
- creare il cursore.
- eseguire "query" SQL (SELECT, INSERT, UPDATE, ecc.) - "query" intende un'operazione SQL sul DB (non solo la select)
- leggere i risultati dal cursore con vari possibili comandi *fetch*: `fetchone()`, `fetchall()` o scorrendo con un ciclo `for`.

Installiamo ora il package `pyodbc` con `pip` (preferibile a `conda` come gestore di package perchè più veloce).

In caso di blocco della esecuzione della cella: da menù `Kernel --> Interrupt Kernel` che interrompe l'esecuzione della cella correntemente NON chiude il kernel.

```
In [5]: !pip install pyodbc
```

```
Requirement already satisfied: pyodbc in c:\users\utente\anaconda3\lib\site-packages (5.2.0)
```

```
In [6]: !pip show pyodbc
```

```
Name: pyodbc
Version: 5.2.0
Summary: DB API module for ODBC
Home-page: https://github.com/mkleehammer/pyodbc
Author:
Author-email: Michael Kleehammer <michael@kleehammer.com>
License: MIT-0 License
Location: C:\Users\Utente\anaconda3\Lib\site-packages
Requires:
Required-by:
```

```
In [7]: import pyodbc
```

Creiamo un oggetto di classe *connessione pyodbc* (istanziamento):

```
In [8]: conn = pyodbc.connect(
    'DRIVER={ODBC Driver 17 for SQL Server};'
    'SERVER=localhost;'
    'DATABASE=ordini_clienti;' # <-- il DB creato da SSMS
    'Trusted_Connection=yes;'
)
```

```
In [9]: type(conn)
```

```
Out[9]: pyodbc.Connection
```

Ora usiamo un **cursore** (ne spiegheremo il significato tra breve):

```
In [11]: cursor = conn.cursor()
```

```
In [12]: type(cursor)
```

```
Out[12]: pyodbc.Cursor
```

Un *cursore* è costituito da:

- una **memoria interna** (*buffer*), composta di varie righe, **temporanea**
- un **puntatore** alla memoria (cioè ad una determinata riga), mi dice quale è la riga corrente. Dopo il comando `execute` il puntatore è posizionato sulla PRIMA riga dei risultati.

Il cursore è acceduto (letto) tramite le istruzioni *fetch*, che spostano anche il puntatore per le righe lette.

Questa framework NON è propria solo di SQLServer ma di tutti gli ambienti DB relazionali, acceduti via Python, che rispettano lo standard ODBC e la Python Database API Specification v2.0.

Una volta attivata la connessione e il cursore, possiamo eseguire la nostra prima SELECT (di prova, estrae semplicemente la data corrente).

```
In [13]: cursor.execute("SELECT GETDATE();")
```

```
Out[13]: <pyodbc.Cursor at 0x22de5ebe030>
```

`<pyodbc.Cursor at 0x1e027204eb0` è il messaggio CORRETTO, NON è un errore! Ci fornisce semplicemente la posizione del cursore.

La select è stata eseguita correttamente.

Non resta che andare a prendere (*fetch*) le righe del risultato e visualizzarle, tramite un ciclo `for` sul cursore:

```
In [14]: for row in cursor:
          print(row)
conn.close()
```

```
(datetime.datetime(2025, 10, 17, 14, 57, 17, 713000),)
```

Vediamo meglio cosa è un *cursore* con un esempio di select più significativo (cioè che accede al database `ordini_clienti`). Prima dobbiamo ristabilire la connessione, chiusa prima.

```
In [20]: conn = pyodbc.connect(
          'DRIVER={ODBC Driver 17 for SQL Server};'
          'SERVER=localhost;'
          'DATABASE=ordini_clienti;'      # <-- il DB creato da SSMS
          'Trusted_Connection=yes;'
        )
```

La seguente cella implementa la *Python Database API Specification v2.0*:

```
In [21]: # 1. Cursore:
# È l'oggetto che ti permette di eseguire comandi SQL (SELECT, INSERT, ecc.)
# e di navigare tra i risultati riga per riga.
cursor = conn.cursor()

# 2. Esegui una query
cursor.execute("SELECT ClienteID, Nome FROM Clienti")

# 3. Fetch: recupera i dati dal buffer dal cursore
# fetchone() -> una riga (quella corrente, cioè quella puntata dal puntatore del
# fetchall() -> tutte le righe della memoria del cursore, indipendentemente dall
# fetchmany(n) -> n righe (le prossime n righe, cioè una lettura a BLOCCHI, a pa

# Esempio con fetchall()
rows = cursor.fetchall() # tutte le righe Le
print(type(rows))        # la classe dell'og

# i dati grezzi del cursore
rows
```

<class 'list'>

```
Out[21]: [(1, 'Antonio Bianchi'),
(2, 'Laura Verdi'),
(3, 'Marco Rossi'),
(4, 'Giulia Neri'),
(5, 'Davide Gialli'),
(6, 'Sara Blu'),
(7, 'Francesco Rosa'),
(8, 'Elena Viola'),
(9, 'Luca Marrone'),
(10, 'Chiara Grigi')]
```

```
In [22]: # Loop di fetch:
for row in rows: # i nomi delle vari
                # sono a scelta del

    print(f"ClienteID: {row.ClienteID}, Nome: {row.Nome}") # questo particolare
                                                         # l'ID e il nome de

# Chiudi la connessione
conn.close()
```

```
ClienteID: 1, Nome: Antonio Bianchi
ClienteID: 2, Nome: Laura Verdi
ClienteID: 3, Nome: Marco Rossi
ClienteID: 4, Nome: Giulia Neri
ClienteID: 5, Nome: Davide Gialli
ClienteID: 6, Nome: Sara Blu
ClienteID: 7, Nome: Francesco Rosa
ClienteID: 8, Nome: Elena Viola
ClienteID: 9, Nome: Luca Marrone
ClienteID: 10, Nome: Chiara Grigi
```

Nota sulla `print` della cella precedente (mista: testo fisso e variabili):

 1. **Stile classico con virgole:**

```
print("Cliente ID: ", row.ClienteID, "Nome: ", row.Nome)
```

✓ Vantaggi:

- Semplice e diretto
- Puoi passare più argomenti separati da virgole (Python li converte in stringhe)
- Aggiunge automaticamente spazi tra gli argomenti

⚠ Svantaggi:

- Meno controllo su formattazione fine (es. niente padding, zeri, decimali)
- Meno elegante per output strutturato
- Spazio automatico potrebbe non essere desiderato

2. **Stile f-string (introdotto in Python 3.6):**

```
print(f"ClienteID: {row.ClienteID}, Nome: {row.Nome}")
```

✓ Vantaggi:

- Più leggibile e compatto
- Più controllo su come formatti i dati

```
f"{prezzo:.2f}"      # due decimali
f"{id:05d}"          # padding con zeri``
```

- È il modo consigliato oggi per formattare stringhe dinamiche

⚠ Svantaggi:

- Funziona solo in Python ≥ 3.6

Un'alternativa: sqlalchemy

Un'alternativa più moderna è: sqlalchemy pymysql .

Cos'è SQLAlchemy?

SQLAlchemy è una libreria più avanzata e flessibile (di pyodbc) per interagire con i database (in Python).

SQLAlchemy è una libreria Python per la gestione dei **database relazionali**. È una delle più potenti e flessibili nel panorama Python. Ti permette di interagire con database come **SQLite, PostgreSQL, MySQL, SQL Server, Oracle**, ecc., in modo astratto e portabile, senza dover scrivere SQL "a mano" (ma se vuoi, puoi farlo lo stesso!).

Vantaggi:

- Multidatabase (supporta SQL Server, PostgreSQL, SQLite, MySQL, ecc.)
- Consente di usare SQL se vuoi, ma anche di evitarlo, grazie al sistema ORM maturo e potente
- Facile da integrare in app web o in pipeline di data science / ML
- Ottimo per progetti web, ETL, data science, ecc.

- Meno codice ripetitivo

Storia di SQLAlchemy

Il nome "SQLAlchemy" è un gioco di parole che unisce:

- **SQL**: il linguaggio standard per interagire con i database relazionali
- **Alchemy**, ovvero "alchimia", a indicare un processo di trasformazione sofisticato, quasi "magico", tra codice Python e SQL

L'idea dietro il nome è che SQLAlchemy compie un'opera "alchemica", trasformando oggetti Python in comandi SQL (e viceversa), mantenendo controllo, potenza e flessibilità.

Chi ha inventato SQLAlchemy?

SQLAlchemy è stato creato da **Mike Bayer**, un ingegnere software statunitense, nel **2005**. Bayer è tuttora il principale manutentore del progetto e un punto di riferimento nella comunità Python per quanto riguarda il data access.

Curiosità:

- SQLAlchemy è **un progetto open source molto maturo e ampiamente utilizzato in ambito professionale**.
- Viene spesso scelto per la sua architettura a strati e per la libertà che offre nello scrivere codice Python interfacciato con il database, con o senza ORM.

Installazione di SQLAlchemy

Il seguente comando va eseguito nel terminale o prompt dei comandi (oppure nel file Python preceduto da !).

Serve per installare le due librerie che ti permettono di connetterti a SQL Server. Vedi più avanti per altri dettagli su queste due librerie.

```
In [14]: !pip install sqlalchemy pyodbc
```

Channels:

- defaults

Platform: win-64

Collecting package metadata (repodata.json): ...working... done

Solving environment: ...working... done

All requested packages already installed.

Nota su package manager

Il gestore di package *conda*, come tutti i gestori di package (anche il pip o uv), cerca il package da installare su vari canali (*channels*). Per package secondari, può succedere che un gestore trovi il package richiesto (da installare) sui propri canali ed un altro gestore invece no (perchè usa altri canali).

In generale, è bene usare dentro un singolo notebook un solo gestore di package (conda

è quello preferibile).

L'installazione del package con conda è stata eseguita correttamente se al termine del messaggio di output di conda trovate le seguenti 3 righe:

Preparing transaction: done

Verifying transaction: done

Executing transaction: done

sqlalchemy vs pyodbc

sqlalchemy : una libreria che fornisce **un'interfaccia ad alto livello** per lavorare con database SQL in Python.

pyodbc : il **driver** che permette a SQLAlchemy (o Python in generale) di connettersi a un database Microsoft **SQL Server**

L'engine di SQLAlchemy

Ora dobbiamo creare l'**engine di SQLAlchemy**, una componente specifica di sqlalchemy.

In SQLAlchemy, l'engine è il **componente centrale** che gestisce la connessione al database e traduce le operazioni SQLAlchemy (Core oppure ORM) in istruzioni SQL eseguibili.

In parole semplici:

L'engine è il "motore" che collega il codice Python a un database (PostgreSQL, SQLite, MySQL, ecc.). È il punto di accesso attraverso cui vengono inviate query, comandi di inserimento, aggiornamento e così via.

Più in dettaglio: ecco cosa fa l'engine:

- Gestisce la connessione al database (usando un connection pool).
- Compila e invia le istruzioni SQL.
- Restituisce i risultati.
- Coordina le transazioni (begin, commit, rollback).
- Si interfaccia con il dialetto SQL corretto (ogni database ha lievi differenze nel linguaggio SQL).

La creazione ("istanziamento") dell'engine richiede la stringa di connessione.

La pila di connessione

Costruzione della **stringa di connessione**:

```
In [25]: # Usiamo ODBC con Trusted_Connection per l'autenticazione Windows.
connection_string = (
    "mssql+pyodbc://localhost/ordini_clienti"
    "?driver=ODBC+Driver+17+for+SQL+Server"
    "&trusted_connection=yes"
)
```


La costruzione di questa stringa di connessione è ****differente da pyodbc e dalla *Python Database API Specification v2.0****.

Essa è tipica di sqlalchemy.

È una stringa che definisce il modo in cui SQLAlchemy si connette al database.

Vediamo la stringa passo a passo:

```
dialect+driver://user:password@host:port/database?  
option1=value1&option2=value2
```

Nel nostro caso:

```
mssql+pyodbc://localhost/ordini_clienti?  
driver=ODBC+Driver+17+for+SQL+Server&trusted_connection=yes
```

Spiegazione dei componenti:

- `mssql+pyodbc` : *dialect: vuoi connetterti a SQL Server (mssql) usando il driver pyodbc*
- `localhost` : *il nome del server SQL (nel tuo caso è sulla tua macchina)*
- `/ordini_clienti` : **il nome del database** - il nostro database SQLServer (che qualcuno ha creato in qualche modo su SQLServer)
- `?driver=...` : *specifica il driver ODBC da usare (come faresti in pyodbc)*
- `&trusted_connection=yes` : *dice di usare autenticazione Windows (SSPI) anziché username/password*

In altre parole:

SQLAlchemy → usa pyodbc → che a sua volta chiama il driver ODBC Microsoft (`ODBC+Driver+17+for+SQL+Server`) → che si connette a SQL Server. `pyodbc` infatti è solo un **"ponte"** tra Python e i driver ODBC di sistema. Quindi serve anche un driver ODBC installato a livello di sistema operativo.

In ****pyodbc**** puro* avremmo scritto così (vedi sezione su *pyodbc*):

```
import pyodbc
```

```
conn = pyodbc.connect(  
    "Driver={ODBC Driver 17 for SQL Server};"  
    "Server=localhost;"  
    "Database=ordini_clienti;"  
    "Trusted_Connection=yes;"  
)
```

La figura seguente illustra la **pila di connessione completa**:

Un driver **alternativo** a `pyodbc` è `pymssql`, un driver nativo per connettere Python a Microsoft SQL Server.

Tuttavia oggi è **sconsigliato** perchè:

- è meno aggiornato (ultimo rilascio: 2020)
- quindi non supporta tutte le funzionalità moderne di SQL Server
- è meno stabile e mantenuto

- su Linux (nel caso) è più semplice da configurare rispetto a pyodbc + driver ODBC Microsoft

Ora, dopo aver effettuato la connessione al DB, possiamo **creare l'engine**:

```
In [26]: from sqlalchemy import create_engine, text
```

```
In [27]: engine = create_engine(connection_string)
```

```
In [28]: type(engine)
```

```
Out[28]: sqlalchemy.engine.base.Engine
```

Dove sta l'engine?

Non è un server, né un servizio a parte.

È un oggetto Python in memoria, istanziato dalla `create_engine`.

Più precisamente:

- il codice dell'Engine fa parte della libreria sqlalchemy, scritta in Python. **È nel tuo ambiente virtuale o sistema Python (kernel).**
- il database vero e proprio? **È esterno all'Engine**: può essere su disco, su rete, in cloud, ecc.

```
In [29]: %who
```

IFrame	IN_COLAB	Image	conn	connection_string	create_engine
cursor	dataframe_columns		dataframe_hash		
display	dtypes_str	engine	get_dataframes	getpass	hashlib
import_pandas_safely		is_data_frame	json		
os	pyodbc	row	rows	text	

L'engine di SQLAlchemy ci permette di:

- sfruttare funzionalità avanzate come pooling, transazioni, **compatibilità con Pandas** (`read_sql`), ecc.
- passare da un database a un altro con poche modifiche (es. da SQLServer a Postgres)

Il bello di SQLAlchemy è che **astrae il codice Python dal database sottostante**.

Quindi cambiando solo la connection string, possiamo collegarci a Postgres anziché SQL Server, e il continuo a funzionare, a patto che:

- si usano costrutti compatibili (niente `TOP N`, ma `LIMIT`)
- si evitano funzioni o sintassi specifiche di un RDBMS (tipo `GETDATE()` in T-SQL)
- si evitano stored procedure o trigger scritti nel dialetto specifico

Sia *SQLAlchemy* che *pyodbc* forniscono astrazione rispetto al DB sottostante. Per le differenze vedi [questa chat](#).

Le due modalità di utilizzo di *sqlalchemy*

SQLAlchemy fornisce due differenti modalità di interazione con il DB: **Core SQL** e **ORM** (Object Relational Mapper).

Entrambi usano lo stesso "motore" sottostante (**SQLAlchemy Engine**), ma forniscono **livelli di astrazione differenti**. Vedi [qui](#) per un buon confronto tra Core SQL e ORM.

In questo notebook **usiamo la modalità Core SQL**, nella quale scriviamo le query SQL classiche con `select()`, `insert`, ecc, ... e poi le eseguiamo con `conn.execute()`.

Una query complessa

Query complessa sul db `ordini_clienti`: *Per ogni cliente, mostrare quanti prodotti ha acquistato, il totale speso, e i prodotti ordinati, ordinando i risultati per cliente e importo speso (decrescente).*

```
In [30]: # definiamo l'oggetto 'query' (questo nome è solo una CONVENZIONE)
query = '''
SELECT
    c.ClienteID,
    c.Nome AS NomeCliente,
    p.Nome AS Prodotto,
    SUM(do.Quantita) AS TotaleQuantita,
    SUM(do.Quantita * do.PrezzoUnitario) AS TotaleSpeso
FROM Ordini o
JOIN Clienti c ON o.ClienteID = c.ClienteID
JOIN DettagliOrdine do ON o.OrdineID = do.OrdineID
JOIN Prodotti p ON do.ProdottoID = p.ProdottoID
GROUP BY c.ClienteID, c.Nome, p.Nome
ORDER BY c.ClienteID, TotaleSpeso DESC;
'''
```

Cosa fa questa query?

Usa 4 tabelle: `Clienti`, `Ordini`, `DettagliOrdine`, `Prodotti`.

Per ogni cliente e prodotto:

- conta il totale acquistato (`SUM(Quantita)`)
- calcola il totale speso (`Quantita * PrezzoUnitario`)
- ordina per cliente e spesa decrescente

Le triple virgolette `'''` permettono di scrivere testo su più righe senza dover concatenare le stringhe o usare `\n` manualmente.

Esempio equivalente (meno leggibile):

Esempio equivalente (meno leggibile):

```
query = "SELECT c.ClienteID, c.Nome FROM Clienti c"
```

Contro:

```
query = "SELECT c.ClienteID,\n" \
        "        c.Nome\n" \
        "FROM Clienti"
```

Molto più scomodo!

In sintesi:

- `query = ''' ... '''` è solo una stringa contenente SQL
- serve per scrivere blocchi SQL ordinati e leggibili
- può essere poi eseguita con la `conn.execute` DENTRO l'engine
- assicurarsi che non ci siano spazi all'inizio delle righe, che cioè devono essere "left-justified", oppure scriverle come stringa singola se si preferisce.

Per eseguire la query in SQLAlchemy non si usa `cursor.execute()` direttamente come faremmo con pyodbc puro o sqlite3. Si fa invece così, con il metodo `execute` della `conn` [dove l'istruzione python `with` serve a stabilire il **contesto** per il blocco di codice successivo alla `width` . Vedi anche [qui](#)]:

```
In [31]: with engine.connect() as conn:
         result = conn.execute(text(query))
         for row in result:
             print(row)
```

```
(1, 'Antonio Bianchi', 'Notebook Dell Inspiron', 5, Decimal('3745.00'))
(1, 'Antonio Bianchi', 'Stampante HP LaserJet', 5, Decimal('645.00'))
(1, 'Antonio Bianchi', 'Tastiera Meccanica Corsair', 5, Decimal('445.00'))
(1, 'Antonio Bianchi', 'Webcam Logitech C920', 4, Decimal('359.60'))
(1, 'Antonio Bianchi', 'Smartwatch Xiaomi Mi Band 7', 3, Decimal('149.70'))
(2, 'Laura Verdi', 'Speaker Bluetooth JBL', 6, Decimal('359.40'))
(2, 'Laura Verdi', 'Smartwatch Xiaomi Mi Band 7', 7, Decimal('349.30'))
(2, 'Laura Verdi', 'Monitor LG UltraFine 27"', 1, Decimal('299.00'))
(3, 'Marco Rossi', 'Notebook Lenovo IdeaPad', 3, Decimal('1797.00'))
(3, 'Marco Rossi', 'Tablet Samsung Galaxy Tab A8', 3, Decimal('687.00'))
(3, 'Marco Rossi', 'Webcam Logitech C920', 7, Decimal('629.30'))
(3, 'Marco Rossi', 'SSD Kingston 500GB', 9, Decimal('539.10'))
(3, 'Marco Rossi', 'Monitor 24" Samsung', 1, Decimal('149.90'))
(3, 'Marco Rossi', 'Hard Disk Esterno WD 1TB', 2, Decimal('139.80'))
(3, 'Marco Rossi', 'Stampante HP LaserJet', 1, Decimal('129.00'))
(3, 'Marco Rossi', 'Smartwatch Xiaomi Mi Band 7', 1, Decimal('49.90'))
(3, 'Marco Rossi', 'Router TP-Link Archer', 1, Decimal('39.90'))
(3, 'Marco Rossi', 'Caricatore USB Aukey', 1, Decimal('19.00'))
(4, 'Giulia Neri', 'Monitor 24" Samsung', 5, Decimal('749.50'))
(4, 'Giulia Neri', 'Stampante HP LaserJet', 4, Decimal('516.00'))
(4, 'Giulia Neri', 'SSD Kingston 500GB', 1, Decimal('59.90'))
(5, 'Davide Gialli', 'Monitor LG UltraFine 27"', 5, Decimal('1495.00'))
(5, 'Davide Gialli', 'Hard Disk Esterno WD 1TB', 5, Decimal('349.50'))
(5, 'Davide Gialli', 'Smartwatch Xiaomi Mi Band 7', 2, Decimal('99.80'))
(5, 'Davide Gialli', 'Pen Drive Sandisk 64GB', 2, Decimal('25.80'))
(6, 'Sara Blu', 'Notebook Lenovo IdeaPad', 2, Decimal('1198.00'))
(6, 'Sara Blu', 'Cuffie Sony WH-1000XM4', 4, Decimal('1116.00'))
(6, 'Sara Blu', 'Hard Disk Esterno WD 1TB', 9, Decimal('629.10'))
(6, 'Sara Blu', 'Tastiera Meccanica Corsair', 3, Decimal('267.00'))
(6, 'Sara Blu', 'SSD Kingston 500GB', 3, Decimal('179.70'))
(7, 'Francesco Rosa', 'Notebook Dell Inspiron', 8, Decimal('5992.00'))
(7, 'Francesco Rosa', 'Tablet Samsung Galaxy Tab A8', 1, Decimal('229.00'))
(7, 'Francesco Rosa', 'Monitor 24" Samsung', 1, Decimal('149.90'))
(7, 'Francesco Rosa', 'Router TP-Link Archer', 2, Decimal('79.80'))
(7, 'Francesco Rosa', 'Mouse Logitech M185', 2, Decimal('39.98'))
(8, 'Elena Viola', 'Webcam Logitech C920', 2, Decimal('179.80'))
(8, 'Elena Viola', 'Router TP-Link Archer', 4, Decimal('159.60'))
(8, 'Elena Viola', 'Mouse Logitech M185', 4, Decimal('79.96'))
(8, 'Elena Viola', 'Pen Drive Sandisk 64GB', 4, Decimal('51.60'))
(8, 'Elena Viola', 'Caricatore USB Aukey', 1, Decimal('19.00'))
(9, 'Luca Marrone', 'Mouse Logitech M185', 5, Decimal('99.95'))
(9, 'Luca Marrone', 'Altoparlanti Logitech Z200', 2, Decimal('69.80'))
(10, 'Chiara Grigi', 'Tastiera Meccanica Corsair', 4, Decimal('356.00'))
(10, 'Chiara Grigi', 'Speaker Bluetooth JBL', 5, Decimal('299.50'))
(10, 'Chiara Grigi', 'Webcam Logitech C920', 2, Decimal('179.80'))
(10, 'Chiara Grigi', 'Smartwatch Xiaomi Mi Band 7', 3, Decimal('149.70'))
(10, 'Chiara Grigi', 'Cartucce HP 302', 5, Decimal('124.50'))
(10, 'Chiara Grigi', 'SSD Kingston 500GB', 2, Decimal('119.80'))
(10, 'Chiara Grigi', 'Mouse Logitech M185', 1, Decimal('19.99'))
```

Cosa fa la cella precedente?

1. `with engine.connect() as conn:`

Crea una **connessione al database** utilizzando l'oggetto engine di SQLAlchemy creato precedentemente.

Il blocco `with` garantisce che la connessione venga chiusa automaticamente alla fine del blocco, anche se si verifica un errore.

`conn` è l'oggetto che usiamo per inviare comandi SQL (l'avevamo usata anche con `pyodbc`).

2. `result = conn.execute(text(query))`

Esegue la query SQL sul database.

`text(query)` è un wrapper sicuro fornito da SQLAlchemy per scrivere query raw SQL (cioè stringhe di SQL classico):

- crea un oggetto `TextClause`, che rappresenta una query SQL "raw".
- permette di usare parametri (`:nome`) stile SQLAlchemy, più sicuri delle f-string.
- è utile quando non vuoi (o non puoi) usare ORM o SQLAlchemy Core.

`result` è un oggetto iterabile che rappresenta il set di risultati restituito dalla query.

3. `for row in result:`

Itera su tutte le righe del risultato (`result`).

Ogni row è un oggetto simile a una tupla o un dict-like (a seconda del tipo di query).

4. `print(row)`

Stampa la riga.

Punto chiave: non si lavora più con il cursore (come in `pyodbc`) ma con la stringa di connessione.

Ora possiamo eseguire una query SQL sul nostro DB e caricare i risultati in pandas (GRANDE VANTAGGIO) con la funzione `pd.read_sql` :

```
In [32]: import pandas as pd
with engine.connect() as conn:
    df = pd.read_sql("SELECT * FROM Clienti", conn)    # argomenti in input: La
    display(df.head(10))
```

	ClientelD	Nome	Email	Telefono	DataRegistrazione
0	1	Antonio Bianchi	antonio.bianchi@email.com	3331234567	2025-06-05
1	2	Laura Verdi	laura.verdi@email.com	3332345678	2025-06-05
2	3	Marco Rossi	marco.rossi@email.com	3333456789	2025-06-05
3	4	Giulia Neri	giulia.neri@email.com	3334567890	2025-06-05
4	5	Davide Gialli	davide.gialli@email.com	3335678901	2025-06-05
5	6	Sara Blu	sara.blu@email.com	3336789012	2025-06-05
6	7	Francesco Rosa	francesco.rosa@email.com	3337890123	2025-06-05
7	8	Elena Viola	elena.viola@email.com	3338901234	2025-06-05
8	9	Luca Marrone	luca.marrone@email.com	3339012345	2025-06-05
9	10	Chiara Grigi	chiara.grigi@email.com	3330123456	2025-06-05

```
In [33]: df.shape
```

```
Out[33]: (10, 5)
```

Proviamo ora con una *****select*** più complessa** ([quella di prima](#) con i join, le group-by e le order-by), strutturata per chiarezza su **diverse righe**. Vedi [qui](#) alcune note su come strutturare le istruzioni python su più righe (modi giusti e modi sbagliati).

```
In [39]: with engine.connect() as conn:
          df = pd.read_sql(
              """SELECT
                  c.ClienteID,
                  c.Nome AS NomeCliente,
                  p.Nome AS Prodotto,
                  SUM(do.Quantita) AS TotaleQuantita,
                  SUM(do.Quantita * do.PrezzoUnitario) AS TotaleSpeso
              FROM Ordini o
              JOIN Clienti c ON o.ClienteID = c.ClienteID
              JOIN DettagliOrdine do ON o.OrdineID = do.OrdineID
              JOIN Prodotti p ON do.ProdottoID = p.ProdottoID
              GROUP BY c.ClienteID, c.Nome, p.Nome
              ORDER BY c.ClienteID, TotaleSpeso DESC""",
              conn)      # argomenti in input: la stringa e l'oggetto connessione

          display(df.head(10))
```

	ClientelD	NomeCliente	Prodotto	TotaleQuantita	TotaleSpeso
0	1	Antonio Bianchi	Notebook Dell Inspiron	5	3745.0
1	1	Antonio Bianchi	Stampante HP LaserJet	5	645.0
2	1	Antonio Bianchi	Tastiera Meccanica Corsair	5	445.0
3	1	Antonio Bianchi	Webcam Logitech C920	4	359.6
4	1	Antonio Bianchi	Smartwatch Xiaomi Mi Band 7	3	149.7
5	2	Laura Verdi	Speaker Bluetooth JBL	6	359.4
6	2	Laura Verdi	Smartwatch Xiaomi Mi Band 7	7	349.3
7	2	Laura Verdi	Monitor LG UltraFine 27"	1	299.0
8	3	Marco Rossi	Notebook Lenovo IdeaPad	3	1797.0
9	3	Marco Rossi	Tablet Samsung Galaxy Tab A8	3	687.0

Domande e risposte di chatGPT

Domanda 1: l'oggetto `conn` in `sqlalchemy`, usato da `engine.connect` ha un proprietà o attributo che restituisca la stringa di connessione usata?

No, non ha direttamente un attributo che restituisce la stringa di connessione usata, ma puoi risalire tramite il suo engine, in questo modo (generico, fornito da chatGPT):

```
In [ ]: from sqlalchemy import create_engine

engine = create_engine("sqlite:///miofile.db")
conn = engine.connect()

# Recupera la stringa di connessione
connection_string = str(conn.engine.url)

print(connection_string)
```

Il codice precedente personalizzato al nostro caso:

```
In [22]: connection_string = (
        "mssql+pyodbc://localhost/ordini_clienti"
        "?driver=ODBC+Driver+17+for+SQL+Server"
        "&trusted_connection=yes"
    )
engine = create_engine(connection_string)
conn = engine.connect()

# Recupera la stringa di connessione
connection_string = str(conn.engine.url)  # 'str' converte un oggetto di tipo U
                                          # 'str' è un data-type di python (non

print(connection_string)
```

mssql+pyodbc://localhost/ordini_clienti?driver=ODBC+Driver+17+for+SQL+Server&trusted_connection=yes

```
In [23]: type(conn.engine.url)
```

```
Out[23]: sqlalchemy.engine.url.URL
```

Domanda 2: spiega la sintassi della stringa di connessione e perchè non si usano le virgole? Ecco [qui](#) la risposta di chatGPT.

Le connessioni in pyodbc

E' bene usare delle **buone pratiche di programmazione** che chiudano la connessione al termine delle operazioni principali, ad esempio con questa classica struttura python

`try ... except ... finally` (da PERSONALIZZARE come stringa di connessione):

```
In [24]: from sqlalchemy import create_engine, text
        from sqlalchemy.exc import SQLAlchemyError

        # Stringa di connessione (es. SQLite per semplicità, ma puoi usare anche Postgre
engine = create_engine('sqlite:///mio_database.db')

conn = None
```



```

try:
    # Crea una connessione
    conn = engine.connect()

    # Esegui una query di esempio
    result = conn.execute(text("SELECT sqlite_version();"))

    # Stampa i risultati
    for row in result:
        print("Versione SQLite:", row[0])

except SQLAlchemyError as e:
    # Gestione degli errori SQLAlchemy
    print("Si è verificato un errore:", e)

finally:
    # Chiude la connessione se aperta
    if conn is not None:
        conn.close()
        print("Connessione chiusa.")

```

Versione SQLite: 3.45.3

Connessione chiusa.

Quando parliamo di struttura `try-except-finally` in **Python**, ci riferiamo a un **meccanismo di gestione degli errori** (o *exception handling*) che serve a:

- intercettare eventuali errori (eccezioni) che potrebbero verificarsi in fase di esecuzione;
- gestirli in modo controllato (senza far “esplodere” il programma);
- eseguire sempre delle operazioni finali (come chiudere connessioni, file, o risorse), indipendentemente dal successo o fallimento del blocco principale.

In [25]:

```

try:
    # Blocco di codice che può generare eccezioni
    ...
except NomeEccezione:
    # Questo blocco viene eseguito solo se si verifica l'eccezione specificata
    ...
finally:
    # Questo blocco viene eseguito SEMPRE, sia che ci sia stata un'eccezione o n
    ...

```

Le **conseguenze di (troppe) connessioni al DB aperte**:

✓ 1. Uso delle risorse del DBMS

Ogni connessione aperta occupa una sessione lato database (ad es. SQL Server, Oracle, PostgreSQL).

Il numero massimo di connessioni è limitato e può causare:

- blocchi se esaurisci il limite,
- lentezza generale del DB (troppe sessioni "idle"),
- errori come Too many connections o Connection pool exhausted.

⚠️ 2. Rischio di blocchi e deadlock Connessioni aperte con transazioni non chiuse possono:

- bloccare risorse (lock su righe, tabelle, ecc.),
- causare deadlock se più processi competono per le stesse risorse,
- tenere trigger e vincoli attivi senza che le modifiche siano ancora visibili.

🔄 3. Gestione delle transazioni Se non chiudi esplicitamente una transazione (commit() o rollback()):

- le modifiche non vengono applicate (nel caso di autocommit=False),
- restano "in sospeso" fino alla chiusura della connessione, -potresti vedere dati inconsistenti da altri client.

💣 4. Potenziali memory leak e crash Le connessioni non chiuse possono portare a:

- memory leak lato Python (es. oggetti connessi ancora in RAM),
- problemi di garbage collection se usi ORM pesanti come SQLAlchemy.

🔒 5. Sicurezza e logging Connessioni aperte senza controllo possono:

- lasciare dati sensibili esposti (ad esempio se un errore interrompe il programma),
- creare problemi nei log di auditing (sessioni non terminate correttamente).

✅ Buone pratiche Chiudere sempre la connessione con conn.close() (ODBC) o session.close() (SQLAlchemy).

Usare `with` per garantire chiusura automatica:

```
with engine.connect() as conn:
    result = conn.execute(...)
```

Le connessioni aperte

Il problema pratico: se sono comunque rimaste aperte delle connessioni:

```
In [26]: conn = pyodbc.connect(
          'DRIVER={ODBC Driver 17 for SQL Server};'
          'SERVER=localhost;'
          'DATABASE=ordini_clienti;'    # <-- il DB creato da SSMS
          'Trusted_Connection=yes;'
        )
          cursor = conn.cursor()
```

```
In [27]: # La Lista complessiva (su tutti i DB) delle CONNESSIONI:
          query = '''
          SELECT
              s.session_id,
              s.login_name,
              s.host_name,
              s.program_name,
              c.client_net_address,
              s.status,
              s.cpu_time,
```

```

        s.memory_usage,
        s.login_time,
        s.last_request_start_time,
        c.connect_time,
        c.num_reads AS reads,
        c.num_writes AS writes,
        st.text AS query_text
    FROM sys.dm_exec_sessions s
    JOIN sys.dm_exec_connections c ON s.session_id = c.session_id
    CROSS APPLY sys.dm_exec_sql_text(c.most_recent_sql_handle) AS st
    ORDER BY s.login_time DESC;
'''

cursor.execute(query)

for row in cursor.fetchall():
    print(row, "\n")

```

```

(63, 'DESKTOP-5GVTEG7\\Utente', 'DESKTOP-5GVTEG7', 'Python', '<local machine>',
'running', 0, 3, datetime.datetime(2025, 6, 12, 18, 54, 14, 133000), datetime.datetime(2025, 6, 12, 18, 54, 16, 793000), datetime.datetime(2025, 6, 12, 18, 54, 14, 130000), 7, 6, '\nSELECT \n      s.session_id,\n      s.login_name,\n      s.host_name,\n      s.program_name,\n      c.client_net_address,\n      s.status,\n      s.cpu_time,\n      s.memory_usage,\n      s.login_time,\n      s.last_request_start_time,\n      c.connect_time,\n      c.num_reads AS reads,\n      c.num_writes AS writes,\n      st.text AS query_text\nFROM sys.dm_exec_sessions s\nJOIN sys.dm_exec_connections c ON s.session_id = c.session_id\nCROSS APPLY sys.dm_exec_sql_text(c.most_recent_sql_handle) AS st\nORDER BY s.login_time DESC;\n')

```

```

(61, 'DESKTOP-5GVTEG7\\Utente', 'DESKTOP-5GVTEG7', 'Python', '<local machine>',
'sleeping', 0, 3, datetime.datetime(2025, 6, 12, 18, 54, 5, 787000), datetime.datetime(2025, 6, 12, 18, 54, 5, 787000), datetime.datetime(2025, 6, 12, 18, 54, 5, 780000), 13, 13, 'SELECT 1 FROM fn_listextendedproperty(default, default, default, default, default, default, default)')

```

```

(62, 'NT SERVICE\\SQLTELEMETRY', 'DESKTOP-5GVTEG7', 'SQLServerCEIP', '<local machine>',
'sleeping', 0, 4, datetime.datetime(2025, 6, 12, 18, 54, 0, 820000), datetime.datetime(2025, 6, 12, 18, 54, 0, 820000), datetime.datetime(2025, 6, 12, 18, 49, 0, 790000), 13, 79, "if not exists (select * from sys.dm_xe_sessions where name = 'telemetry_xevents')\r\n\alter event session telemetry_xevents on server state=start")

```

```

(57, 'DESKTOP-5GVTEG7\\Utente', 'DESKTOP-5GVTEG7', 'Python', '<local machine>',
'sleeping', 16, 4, datetime.datetime(2025, 6, 12, 18, 53, 49, 937000), datetime.datetime(2025, 6, 12, 18, 53, 57, 440000), datetime.datetime(2025, 6, 12, 18, 53, 49, 933000), 23, 24, 'SELECT * FROM Clienti')

```

```

(51, 'DESKTOP-5GVTEG7\\Utente', 'DESKTOP-5GVTEG7', 'Python', '<local machine>',
'sleeping', 0, 3, datetime.datetime(2025, 6, 12, 18, 53, 24, 803000), datetime.datetime(2025, 6, 12, 18, 53, 28, 430000), datetime.datetime(2025, 6, 12, 18, 53, 11, 377000), 17, 18, 'SELECT ClienteID, Nome FROM Clienti')

```

```

In [28]: # Il totale (su tutti i DB):
query = '''
SELECT COUNT(*) AS NumeroConnessioniAttive
FROM sys.dm_exec_sessions s
JOIN sys.dm_exec_connections c ON s.session_id = c.session_id;
'''

```

```
cursor.execute(query)

for row in cursor.fetchall():
    print(row)
```

(5,)

```
In [29]: # Il totale (solo su 'ordini_clienti'):
query = '''
SELECT COUNT(*) AS ConnessioniOrdiniClienti
FROM sys.dm_exec_sessions s
JOIN sys.dm_exec_connections c ON s.session_id = c.session_id
WHERE s.database_id = DB_ID('ordini_clienti');
'''

cursor.execute(query)

for row in cursor.fetchall():
    print(row, "\n")
```

(4,)

Chiudere una sessione

E per chiudere una sessione? Occorre sapere il suo ID, ad esempio così

```
In [30]: # Il totale (solo su 'ordini_clienti'):
query = '''
SELECT
    s.session_id,
    s.login_name,
    s.host_name,
    s.program_name,
    c.client_net_address,
    s.status,
    s.cpu_time,
    s.memory_usage,
    s.login_time,
    s.last_request_start_time,
    c.connect_time,
    c.num_reads AS reads,
    c.num_writes AS writes,
    st.text AS query_text,
    'KILL ' + CAST(s.session_id AS VARCHAR(10)) AS comando_kill
FROM sys.dm_exec_sessions s
JOIN sys.dm_exec_connections c ON s.session_id = c.session_id
CROSS APPLY sys.dm_exec_sql_text(c.most_recent_sql_handle) AS st
ORDER BY s.login_time DESC;
'''

cursor.execute(query)

for row in cursor.fetchall():
    print(row, "\n")
```

```
(63, 'DESKTOP-5GVTEG7\\Utente', 'DESKTOP-5GVTEG7', 'Python', '<local machine>', 'running', 0, 3, datetime.datetime(2025, 6, 12, 18, 54, 14, 133000), datetime.datetime(2025, 6, 12, 18, 54, 22, 520000), datetime.datetime(2025, 6, 12, 18, 54, 14, 130000), 10, 9, "\nSELECT \n      s.session_id,\n      s.login_name,\n      s.host_name,\n      s.program_name,\n      c.client_net_address,\n      s.status,\n      s.cpu_time,\n      s.memory_usage,\n      s.login_time,\n      s.last_request_start_time,\n      c.connect_time,\n      c.num_reads AS reads,\n      c.num_writes AS writes,\n      st.text AS query_text,\n      'KILL ' + CAST(s.session_id AS VARCHAR(10)) AS comando_kill\nFROM sys.dm_exec_sessions s\nJOIN sys.dm_exec_connections c ON s.session_id = c.session_id\nCROSS APPLY sys.dm_exec_sql_text(c.most_recent_sql_handle) AS st\nORDER BY s.login_time DESC;\n\n", 'KILL 63')
```

```
(61, 'DESKTOP-5GVTEG7\\Utente', 'DESKTOP-5GVTEG7', 'Python', '<local machine>', 'sleeping', 0, 3, datetime.datetime(2025, 6, 12, 18, 54, 5, 787000), datetime.datetime(2025, 6, 12, 18, 54, 5, 787000), datetime.datetime(2025, 6, 12, 18, 54, 5, 780000), 13, 13, 'SELECT 1 FROM fn_listextendedproperty(default, default, default, default, default, default, default, default)', 'KILL 61')
```

```
(62, 'NT SERVICE\\SQLTELEMETRY', 'DESKTOP-5GVTEG7', 'SQLServerCEIP', '<local machine>', 'sleeping', 0, 4, datetime.datetime(2025, 6, 12, 18, 54, 0, 820000), datetime.datetime(2025, 6, 12, 18, 54, 0, 820000), datetime.datetime(2025, 6, 12, 18, 49, 0, 790000), 13, 79, "if not exists (select * from sys.dm_xe_sessions where name = 'telemetry_xevents')\r\n\talter event session telemetry_xevents on server state=start", 'KILL 62')
```

```
(57, 'DESKTOP-5GVTEG7\\Utente', 'DESKTOP-5GVTEG7', 'Python', '<local machine>', 'sleeping', 16, 4, datetime.datetime(2025, 6, 12, 18, 53, 49, 937000), datetime.datetime(2025, 6, 12, 18, 53, 57, 440000), datetime.datetime(2025, 6, 12, 18, 53, 49, 933000), 23, 24, 'SELECT * FROM Clienti', 'KILL 57')
```

```
(51, 'DESKTOP-5GVTEG7\\Utente', 'DESKTOP-5GVTEG7', 'Python', '<local machine>', 'sleeping', 0, 3, datetime.datetime(2025, 6, 12, 18, 53, 24, 803000), datetime.datetime(2025, 6, 12, 18, 53, 28, 430000), datetime.datetime(2025, 6, 12, 18, 53, 11, 377000), 17, 18, 'SELECT ClienteID, Nome FROM Clienti', 'KILL 51')
```


L'ultima colonna mostra direttamente il comando che puoi copiare e incollare per terminare quella connessione.

Esegui i comandi SELECT e KILL da SSMS - è lo strumento ufficiale per interagire graficamente con SQL Server. Usa una nuova finestra di query:

- Apri SSMS
- Connettiti al tuo SQL Server
- Clicca su "Nuova Query" in alto a sinistra
- Incolla lì i comandi T-SQL (SELECT ..., KILL ..., ecc.)
- Premi F5 oppure clicca su Esegui

Non eseguire comandi KILL da Python.

È tecnicamente possibile ma rischioso, e poco pratico. I comandi di amministrazione come KILL si fanno sempre dall'ambiente amministrativo (SSMS).

 **Attenzione.** Evita di terminare la tua stessa sessione, altrimenti lo script si interromperà brutalmente.

Il database attivo deve essere **master** o uno diverso da quello della sessione da terminare.

T-SQL

Cos'è T-SQL?

T-SQL sta per Transact-SQL ed è l'estensione proprietaria di Microsoft SQL Server al linguaggio SQL standard.

Aggiunge costrutti come:

- `BEGIN ... END`
- `DECLARE @variabili`
- `WHILE`
- `TRY...CATCH`
- `PRINT` , `WAITFOR` , `MERGE` , ecc.
- ecc

Il T-SQL può essere eseguito da SSMS, ma anche in un notebook o script Python possiamo scrivere sia SQL puro sia T-SQL avanzato — basta che sia compatibile con SQL Server.

Un esempio di T-SQL avanzato in Python.

- `DECLARE` di una variabile
- ciclo `WHILE`
- gestione degli errori con `TRY...CATCH`
- `PRINT` di messaggi nel log di SQLServer

Non ci sono risultati da fetchare, questo codice esegue logica su SQL Server.

```
In [11]: # Eseguire codice T-SQL avanzato (con DECLARE, WHILE, ecc.) da Python con pyodbc

import pyodbc

# Connessione a SQL Server
conn = pyodbc.connect(
    'DRIVER={ODBC Driver 17 for SQL Server};'
    'SERVER=localhost;'
    'DATABASE=ordini_clienti;'
    'Trusted_Connection=yes;'
)

cursor = conn.cursor()

# Esempio di codice T-SQL con variabili, ciclo WHILE e TRY...CATCH
query = """
BEGIN TRY
    DECLARE @i INT = 1;

    WHILE @i <= 5
    BEGIN
        PRINT 'Esecuzione ciclo numero ' + CAST(@i AS VARCHAR);
    END
END TRY
CATCH
BEGIN
    PRINT 'Errore durante l'esecuzione del ciclo';
END
```

```

        SET @i += 1;
    END

    PRINT 'Tutto eseguito correttamente';
END TRY
BEGIN CATCH
    PRINT 'Errore: ' + ERROR_MESSAGE();
END CATCH;
"""

cursor.execute(query)

conn.commit() # Obbligatorio per blocchi che non restituiscono righe

# Ora accediamo ai messaggi
for message in cursor.messages:
    print("Tutti i messaggi aggregati:\n", message)

conn.close()

```

Tutti i messaggi aggregati:

```
('[01000] (0)', '[Microsoft][ODBC Driver 17 for SQL Server][SQL Server]Esecuzione
e ciclo numero 1')
```

Nota: i messaggi `PRINT` non verranno stampati direttamente nel terminale Python, ma restano nel log interno del motore SQL Server.

Li si può vedere da SSMS.

Vediamo ora un esempio di codice T-SQL che **restituisce risultati** (come una `SELECT` dentro un ciclo)

In [12]: *# Eseguire codice T-SQL avanzato da Python con risultati fetchabili*

```

import pyodbc

# Connessione a SQL Server
conn = pyodbc.connect(
    'DRIVER={ODBC Driver 17 for SQL Server};'
    'SERVER=localhost;'
    'DATABASE=ordini_clienti;'
    'Trusted_Connection=yes;'
)

cursor = conn.cursor()

# Crea una tabella temporanea e inserisce righe con un ciclo WHILE
query = """
IF OBJECT_ID('tempdb..#RisultatiCiclo') IS NOT NULL
    DROP TABLE #RisultatiCiclo;

CREATE TABLE #RisultatiCiclo (Numero INT);

DECLARE @i INT = 1;
WHILE @i <= 5
BEGIN
    INSERT INTO #RisultatiCiclo VALUES (@i);
    SET @i += 1;
END

```

```

SELECT * FROM #RisultatiCiclo;
"""

cursor.execute(query)
rows = cursor.fetchall()

for row in rows:
    print(f"Numero: {row.Numero}")

conn.close()

```

```

-----
ProgrammingError                                Traceback (most recent call last)
Cell In[12], line 33
     16 query = """
     17 IF OBJECT_ID('tempdb..#RisultatiCiclo') IS NOT NULL
     18     DROP TABLE #RisultatiCiclo;
    (...    29 SELECT * FROM #RisultatiCiclo;
     30 """
     32 cursor.execute(query)
--> 33 rows = cursor.fetchall()
     35 for row in rows:
     36     print(f"Numero: {row.Numero}")

ProgrammingError: No results. Previous SQL was not a query.

```

L'interessante problema deriva dal fatto che `pyodbc` non consente di fare `fetchall()` se l'ultima istruzione della query non è una `SELECT` isolata. Ecco qui sotto lo script aggiornato diviso in due fasi:

- Prima esegue il blocco con `INSERT` e `WHILE`
- Poi esegue un `SELECT` separata per leggere i dati

```

In [13]: # Eseguire codice T-SQL in più fasi da Python per evitare errori fetch

import pyodbc

# Connessione a SQL Server
conn = pyodbc.connect(
    'DRIVER={ODBC Driver 17 for SQL Server};'
    'SERVER=localhost;'
    'DATABASE=ordini_clienti;'
    'Trusted_Connection=yes;'
)

cursor = conn.cursor()

# 1. Creazione e popolamento della tabella temporanea (con il blocco T-SQL)
setup_query = """
IF OBJECT_ID('tempdb..#RisultatiCiclo') IS NOT NULL
    DROP TABLE #RisultatiCiclo;

CREATE TABLE #RisultatiCiclo (Numero INT);

DECLARE @i INT = 1;
WHILE @i <= 5
BEGIN
    INSERT INTO #RisultatiCiclo VALUES (@i);
    SET @i += 1;

```



```
END
"""

cursor.execute(setup_query)
conn.commit() # Commit necessario per completare il blocco

# 2. Esecuzione della SELECT separatamente
cursor.execute("SELECT * FROM #RisultatiCiclo;")
rows = cursor.fetchall()

for row in rows:
    print(f"Numero: {row.Numero}")

conn.close()
```

```
Numero: 1
Numero: 2
Numero: 3
Numero: 4
Numero: 5
```

PostgreSQL

Molto simile come SQL, meno come parte procedurale (Postgres usa una variante del PL-SQL di Oracle).

Vedi [questo confronto](#).

In []:

In []: