Working with Databases in Python 3

Using a Local Relational Database - SQLite

Douglas Starnes
Author / Speaker

@poweredbyaltnet | linktr.ee/douglasstarnes

Overview

Usage scenarios for SQLite

Visual Studio Code extension

Python Standard Library `sqlite3` module

- Connect to a SQLite database
- Execute SQL queries
- Parse the results

Row factories

Build a command line application to manage a cryptocurrency portfolio

- With live prices!

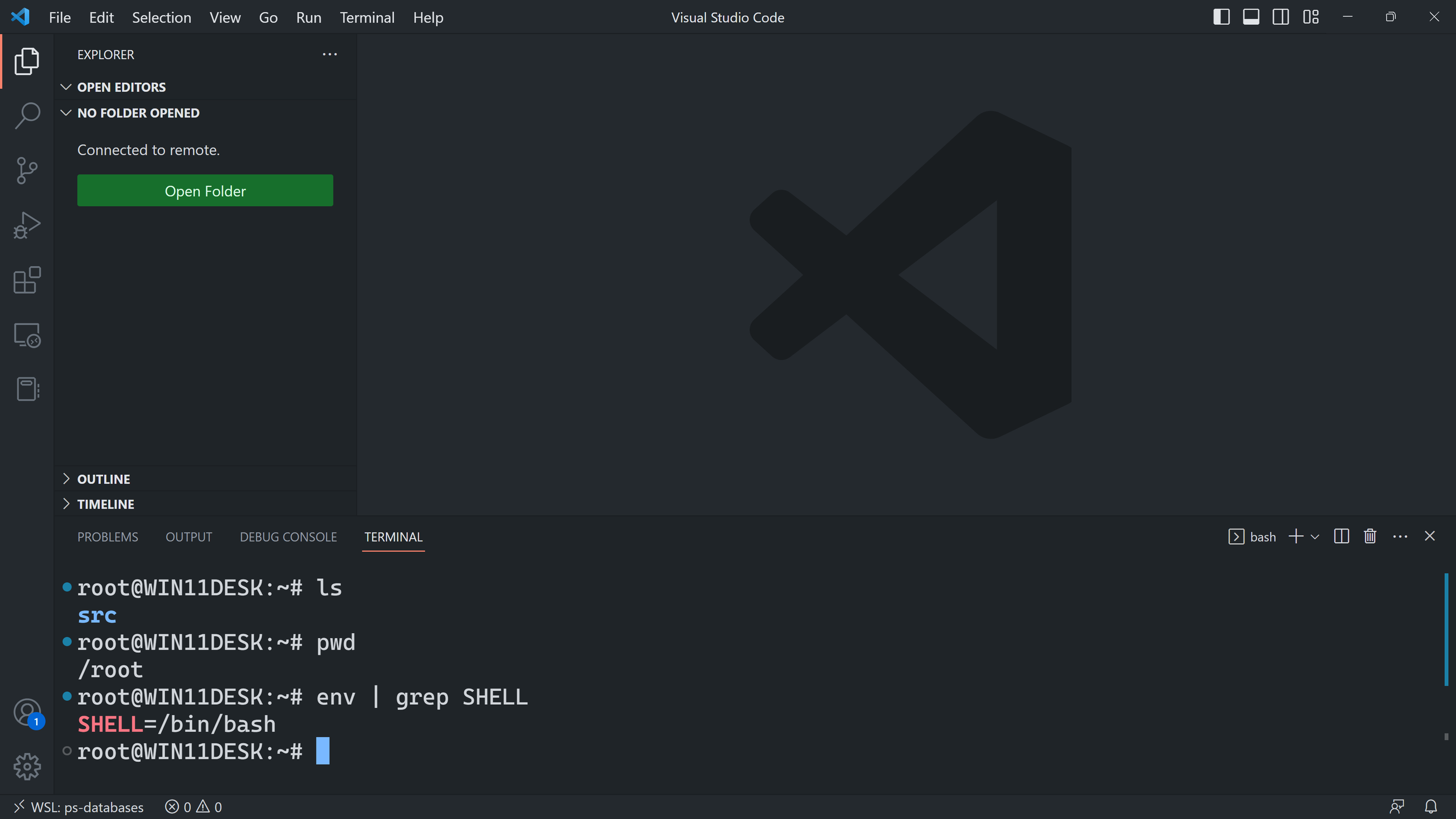
SQLite Usage Scenarios

SQLite does not use a server

Command line client opens a locally stored file

Use as an embedded database

Use during prototyping and development


File Edit Selection View Go Run Terminal Help
Visual Studio Code
EXPLORER
OPEN EDITORS
NO FOLDER OPENED
Connected to remote.
Open Folder
OUTLINE
TIMELINE
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
bash
root@WIN11DESK:~# ls
src
root@WIN11DESK:~# pwd
/root
root@WIN11DESK:~# env | grep SHELL
SHELL=/bin/bash
root@WIN11DESK:~#
WSL: ps-databases
0 0

Demo

Building a command line application

Working with the `sqlite3` Module

```
import sqlite3

database = sqlite3.connect(“portfolio.db”)
```

Working with the `sqlite3` Module

```
import sqlite3

database = sqlite3.connect(“portfolio.db”)

cursor = database.cursor()
```

Working with the `sqlite3` Module

```
import sqlite3

database = sqlite3.connect(“portfolio.db”)

cursor = database.cursor()

create_table_query = “””
CREATE TABLE investments (
    coin_id TEXT,
    currency TEXT,
    sell INT,
    amount REAL,
    date TIMESTAMP
);
“””

cursor.execute(create_table_query)
```

Working with the `sqlite3` Module

```
import datetime

investment = (“bitcoin”, “usd”, True, 1.0, datetime.datetime.now())
```

Working with the `sqlite3` Module

```
import datetime

investment = (“bitcoin”, “usd”, True, 1.0, datetime.datetime.now())

cursor.execute(
    “INSERT INTO investments VALUES (?, ?, ?, ?, ?);”,
    investment
)

database.commit()
```

Working with the `sqlite3` Module

```
import datetime

investment = (“bitcoin”, “usd”, True, 1.0, datetime.datetime.now())

cursor.execute(
    “INSERT INTO investments VALUES (?, ?, ?, ?, ?);”,
    investment
)

database.commit()

result = cursor.execute(“SELECT * FROM investments;”)
```

Working with the `sqlite3` Module

```
import datetime

investment = (“bitcoin”, “usd”, True, 1.0, datetime.datetime.now())

cursor.execute(
    “INSERT INTO investments VALUES (?, ?, ?, ?, ?);”,
    investment
)

database.commit()

result = cursor.execute(“SELECT * FROM investments;”)

all_rows = result.fetchall()
```

Working with the sqlite3 Module

```
import datetime

investment = (“bitcoin”, “usd”, True, 1.0, datetime.datetime.now())

cursor.execute(
    “INSERT INTO investments VALUES (?, ?, ?, ?, ?);”,
    investment
)

database.commit()

result = cursor.execute(“SELECT * FROM investments;”)

all_rows = result.fetchall()

first_or_only_row = result.fetchone()
```

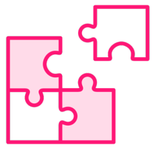
Are We There Yet?

Generic collections such as a list of tuples can be confusing to work with, particularly in a larger application

What if you could use structured Python objects and refer to values by name instead of position?

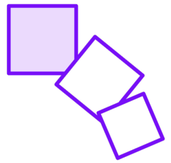
You can!
It’s called a row factory.

Row Factories



The `row_factory` attribute

`sqlite3` includes the `Row` row factory



Transform a tuple into a dictionary like object

You could also use a `namedtuple` from the `collections` module

Dataclasses

```
# Helper class / data transfer object

class Investment:
    def __init__(self, coin_id, currency, amount, sell, date):
        self.coin_id = coin_id
        self.currency = currency
        self.amount = amount
        self.sell = sell
        self.date = date

    def __repr__(self):
        return f"<Investment {self.coin_id} {self.amount:.2f} {self.currency.upper()}>"
```

Dataclasses

```
from dataclasses import dataclass

@dataclass
class Investment:
    coin_id: str
    currency: str
    amount: float
    sell: bool
    date: datetime.datetime
```

Dataclasses

```python
from dataclasses import dataclass

@dataclass
class Investment:
    coin_id: str
    currency: str
    amount: float
    sell: bool
    date: datetime.datetime

def compute_value(self) -> float:
    return self.amount * get_coin_price(self.coin_id, self.currency)
```

Row Factory

```
def investment_row_factory(_, row):
    return Investment(
        coin_id = row[0],
        currency = row[1],
        amount = row[2],
        sell = bool(row[3]),
        date = datetime.datetime.strptime(row[4], “%Y-%m-%d %H:%M:%S.%f”)
    )
```

Row Factory

```
def investment_row_factory(_, row):
    return Investment(
        coin_id = row[0],
        currency = row[1],
        amount = row[2],
        sell = bool(row[3]),
        date = datetime.datetime.strptime(row[4], “%Y-%m-%d %H:%M:%S.%f”)
    )

# ...

database.row_factory = investment_row_factory
```

Summary

SQLite is a file-based database

- sqlite3 command line client
- VS Code extension

`sqlite3` module in the Python Standard Library

Parameterized SQL queries

Results by default are list of tuples

Row factories transform results to Python objects

Build a command line cryptocurrency portfolio manager