Name:Sanika Deshmukh

Domain:Python

Code:

```
import base64
import getpass
import os
import sqlite3
import sys
from dataclasses import dataclass
from datetime import datetime
from secrets import choice
from typing import Iterable, Optional, Sequence

try:
    from cryptography.fernet import Fernet
except ImportError as exc:
    print("The 'cryptography' package is required. Install with: pip install cryptography")
    raise exc

DB_PATH = "vault.db"
ITERATIONS = 390_000
def derive_key(master_password: str, salt: bytes) -> bytes:
    """Derive a Fernet-compatible key from a master password and salt."""
    from hashlib import pbkdf2_hmac

    key = pbkdf2_hmac(
        hash_name="sha256",
        password=master_password.encode("utf-8"),
        salt=salt,
        iterations=ITERATIONS,
```

```
        dklen=32,
    )
    return base64.urlsafe_b64encode(key)


def generate_password(
    length: int = 16,
    use_upper: bool = True,
    use_lower: bool = True,
    use_digits: bool = True,
    use_symbols: bool = True,
) -> str:
    """Generate a strong random password."""
    upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    lower = "abcdefghijklmnopqrstuvwxyz"
    digits = "0123456789"
    symbols = "!@#$%^&*()-_=+[]{};:,.<>/?"

    pools: Sequence[str] = []
    if use_upper:
        pools.append(upper)
    if use_lower:
        pools.append(lower)
    if use_digits:
        pools.append(digits)
    if use_symbols:
        pools.append(symbols)

    if not pools:
        raise ValueError("At least one character set must be enabled")
    if length < len(pools):
        raise ValueError(f"Length must be at least {len(pools)} when all sets are enabled")
```

```
    password_chars = [choice(pool) for pool in pools]
    all_chars = "".join(pools)
    password_chars.extend(choice(all_chars) for _ in range(length - len(password_chars)))

    # Simple shuffle using secrets-compatible randomness
    for i in range(len(password_chars) - 1, 0, -1):
        j = choice(range(i + 1))
        password_chars[i], password_chars[j] = password_chars[j], password_chars[i]

    return "".join(password_chars)

def get_conn() -> sqlite3.Connection:
    conn = sqlite3.connect(DB_PATH)
    conn.execute("PRAGMA foreign_keys = ON;")
    return conn

def init_db(conn: sqlite3.Connection) -> None:
    conn.execute(
        """
        CREATE TABLE IF NOT EXISTS metadata (
            id INTEGER PRIMARY KEY CHECK (id = 1),
            salt BLOB NOT NULL,
            created_at TEXT NOT NULL
        );
        """
    )
    conn.execute(
        """
        CREATE TABLE IF NOT EXISTS entries (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            service TEXT NOT NULL UNIQUE,
```

```
            username TEXT NOT NULL,
            password BLOB NOT NULL,
            created_at TEXT NOT NULL
        );
        """
    )
    conn.commit()

def get_or_create_salt(conn: sqlite3.Connection) -> bytes:
    cur = conn.execute("SELECT salt FROM metadata WHERE id = 1;")
    row = cur.fetchone()
    if row:
        return row[0]

    salt = os.urandom(16)
    conn.execute(
        "INSERT INTO metadata (id, salt, created_at) VALUES (1, ?, ?);",
        (salt, datetime.utcnow().isoformat()),
    )
    conn.commit()
    return salt

def encrypt_password(master_password: str) -> Fernet:
    conn = get_conn()
    init_db(conn)
    salt = get_or_create_salt(conn)
    key = derive_key(master_password, salt)
    return Fernet(key)

@dataclass
class Entry:
```

```
    service: str
    username: str
    password: str
    created_at: str

def add_entry(service: str, username: str, raw_password: str, fernet: Fernet) -> None:
    conn = get_conn()
    init_db(conn)
    encrypted = fernet.encrypt(raw_password.encode("utf-8"))
    conn.execute(
        """
        INSERT INTO entries (service, username, password, created_at)
        VALUES (?, ?, ?, ?)
        ON CONFLICT(service) DO UPDATE SET
            username=excluded.username,
            password=excluded.password;
        """,
        (service, username, encrypted, datetime.utcnow().isoformat()),
    )
    conn.commit()

def get_entry(service: str, fernet: Fernet) -> Optional[Entry]:
    conn = get_conn()
    init_db(conn)
    cur = conn.execute(
        "SELECT service, username, password, created_at FROM entries WHERE service = ?;",
        (service,),
    )
    row = cur.fetchone()
    if not row:
        return None
```

```
    decrypted = fernet.decrypt(row[2]).decode("utf-8")
    return Entry(service=row[0], username=row[1], password=decrypted, created_at=row[3])

def list_services() -> Iterable[str]:
    conn = get_conn()
    init_db(conn)
    cur = conn.execute("SELECT service FROM entries ORDER BY service;")
    return [row[0] for row in cur.fetchall()]

def delete_entry(service: str) -> bool:
    conn = get_conn()
    init_db(conn)
    cur = conn.execute("DELETE FROM entries WHERE service = ?;", (service,))
    conn.commit()
    return cur.rowcount > 0

def prompt_master_password() -> str:
    pw1 = getpass.getpass("Master password: ")
    if not pw1:
        print("Master password cannot be empty.")
        sys.exit(1)
    return pw1

def prompt_service() -> str:
    svc = input("Service name: ").strip()
    if not svc:
        print("Service is required.")
    return svc

def handle_add(fernet: Fernet) -> None:
    service = prompt_service()
```

```
    if not service:
        return
    username = input("Username: ").strip()
    if not username:
        print("Username is required.")
        return

    use_generated = input("Generate strong password? [y/N]: ").strip().lower() == "y"
    if use_generated:
        pw = generate_password()
        print(f"Generated password: {pw}")
    else:
        pw = getpass.getpass("Password: ")
    add_entry(service, username, pw, fernet)
    print(f"Saved entry for {service}.")

def handle_get(fernet: Fernet) -> None:
    service = prompt_service()
    if not service:
        return
    entry = get_entry(service, fernet)
    if not entry:
        print("No entry found.")
        return
    print(f"Service: {entry.service}")
    print(f"Username: {entry.username}")
    print(f"Password: {entry.password}")
    print(f"Created: {entry.created_at}")

def handle_list() -> None:
    services = list_services()
```

```
    if not services:
        print("No entries.")
        return
    print("Stored services:")
    for svc in services:
        print(f"- {svc}")


def handle_delete() -> None:
    service = prompt_service()
    if not service:
        return
    if delete_entry(service):
        print("Deleted.")
    else:
        print("No such entry.")


def main() -> None:
    master_password = prompt_master_password()
    fernet = encrypt_password(master_password)

    actions = {
        "1": ("Add / update password", lambda: handle_add(fernet)),
        "2": ("Retrieve password", lambda: handle_get(fernet)),
        "3": ("List services", handle_list),
        "4": ("Delete entry", handle_delete),
        "5": ("Generate password only", lambda: print(generate_password())),
        "0": ("Exit", None),
    }

    while True:
        print(
```

```
            "\nPassword Manager\n"
            "1) Add/Update\n"
            "2) Retrieve\n"
            "3) List\n"
            "4) Delete\n"
            "5) Generate password\n"
            "0) Exit\n"
        )
        choice_input = input("Select: ").strip()
        action = actions.get(choice_input)
        if not action:
            print("Invalid choice.")
            continue
        if choice_input == "0":
            print("Goodbye.")
            break
        _, func = action
        if func:
            func()

if __name__ == "__main__":
    main()
```

Output:

```
Master password:

Password Manager
1) Add/Update
2) Retrieve
3) List
4) Delete
5) Generate password
0) Exit

Select: 5
e3Z1m!HdHG6n+$dN
```