Python i MySQL

MySQL to jeden z najpopularniejszych systemów zarządzania relacyjnymi bazami danych, często wykorzystywany w aplikacjach webowych, systemach zarządzania informacjami czy analityce danych. Python, dzięki swojej prostocie i bogatej bibliotece, jest idealnym językiem do pracy z bazami danych. Połączenie Pythona z MySQL pozwala na tworzenie aplikacji, które mogą:

- Przechowywać i zarządzać dużymi ilościami danych.
- Wykonywać dynamiczne zapytania do bazy danych.
- Zapewniać spójność i bezpieczeństwo danych w środowiskach produkcyjnych.

Praca z MySQL w Pythonie obejmuje takie zadania jak:

1. Tworzenie baz danych i tabel.
2. Operacje CRUD (Create, Read, Update, Delete).
3. Zarządzanie transakcjami i użytkownikami.
4. Bezpieczne wykonywanie zapytań dzięki parametryzacji.

W praktyce, Python może być wykorzystany zarówno do automatyzacji procesów zarządzania bazą danych, jak i do integracji z większymi aplikacjami, np. webowymi. Dzięki modułom takim jak `mysql-connector-python`, programiści mogą w prosty sposób zarządzać bazami danych oraz tworzyć wydajne i skalowalne aplikacje.

Instalacja biblioteki `mysql-connector-python`

```
pip install mysql-connector-python

Requirement already satisfied: mysql-connector-python in c:\users\
bbrzek\anaconda3\lib\site-packages (9.1.0)
Note: you may need to restart the kernel to use updated packages.
```

Połączenie z serwerem MySQL

```
import mysql.connector

conn = mysql.connector.connect(
    host="localhost",
    user="root",         # Nazwa użytkownika
    password="root"   # Hasło użytkownika
)
```

Kursor

Obiekt w Pythonie używany do wysyłania zapytań SQL do bazy danych i pobierania wyników. Wyniki zapytań są przechowywane w buforze kursora (czyli w obiekcie `cursor`)

```
cursor = conn.cursor()
```

---

1. Tworzenie i zarządzanie bazami danych

- Jak stworzyć nowe bazy danych z poziomu Python.
- Jak zarządzać bazami danych (np. sprawdzanie istniejących baz, ich usuwanie).
- Jak w prosty sposób automatyzować te operacje w aplikacjach.

1.1. Tworzenie bazy danych

- `.execute()` - funkcja używana do wysyłania komend SQL z poziomu programu Python.

```
cursor.execute("CREATE DATABASE IF NOT EXISTS student_database")
print("Baza danych została utworzona!")
```

```
Baza danych została utworzona!
```

Zamknięcie kursora i połączenia

- każde otwarte połączenie z bazą danych wykorzystuje zasoby serwera
- unikanie `memory leaks` i przejęcia otwartego połączenia

```
cursor.close()
conn.close()
```

Alternatywne podejście z użyciem `with`

```
import mysql.connector

with mysql.connector.connect(
    host="localhost",
    user="root",
    password="root",
    database="sakila" # Bezpośrednie połączenie z bazą
) as conn:
    with conn.cursor() as cursor:
        cursor.execute("""SELECT *
                        FROM country
                        LIMIT 3;""")
        for row in cursor:
            print(row)  # Automatyczne zamknięcie kursora i połączenia
```

```
(1, 'Afghanistan', datetime.datetime(2006, 2, 15, 4, 44))
(2, 'Algeria', datetime.datetime(2006, 2, 15, 4, 44))
(3, 'American Samoa', datetime.datetime(2006, 2, 15, 4, 44))
```

1.2. Zarządzanie bazami danych

- `.fetchall()` - funckja pobierająca wszystkie wiersze wyników z zapytania

```
cursor.execute("SHOW DATABASES")
databases = [db[0] for db in cursor.fetchall()]
print("Istniejące bazy danych:")
```

```
for db in databases:
    print(f"- {db}")
```

```
Istniejące bazy danych:
- information_schema
- mysql
- performance_schema
- sakila
- student_database
- sys
- world
```

Zarządzanie bazami z poziomu Pythona, to nie tylko tworzenie, usuwanie i wyświetlanie baz danych. Oprócz tych operacji można również sprawdzać szczegóły konfiguracji baz, kopiować je, zmieniach ich kodowanie, optymalizować ich wydajność i wiele więcej.

---

2. Tworzenie i zarządzanie tabelami

- Jak stworzyć nową tabelę w bazie danych?
- Jak wyświetlić tabele znajdujące się w bazie?
- Jak usunąć tabelę?
- Jak zmienić strukturę tabeli?

```
table_name = "students"
```

2.1. Tworzenie nowej tabeli

```
table_name = "students"
create_table_query = f"""
CREATE TABLE IF NOT EXISTS {table_name} (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
"""
cursor.execute(create_table_query)
print(f"Tabela '{table_name}' została utworzona.")
```

```
Tabela 'students' została utworzona.
```

2.2. Wyświetlenie listy tabel

```
cursor.execute("SHOW TABLES")
tables = [table[0] for table in cursor.fetchall()]
print("Istniejące tabele w bazie danych:")
for table in tables:
```

```
    print(f"- {table}")


Istniejące tabele w bazie danych:
- students
```

2.3. Zmiana struktury tabeli

```
add_column_query = f"ALTER TABLE {table_name} ADD COLUMN email
VARCHAR(100)"
cursor.execute(add_column_query)
print("Kolumna 'email' została dodana do tabeli 'students'.")

Kolumna 'email' została dodana do tabeli 'students'.
```

2.4. Usunięcie tabeli

```
drop_table_query = f"DROP TABLE IF EXISTS {table_name}"
cursor.execute(drop_table_query)
print(f"Tabela '{table_name}' została usunięta.")

Tabela 'students' została usunięta.
```

---

3. Operacje CRUD (Create, Read, Update, Delete)

- INSERT – Dodawanie danych do tabel.
- SELECT – Odczyt danych z tabel.
- UPDATE – Aktualizacja istniejących rekordów.
- DELETE – Usuwanie rekordów z tabel.
- `.executemany()` - wykonuje to samo zapytanie SQL dla wielu zestawów danych (szybsze niż wielokrotne wywołanie `.execute()`)
- `.commit()` - zatwierdza wszystkie zmiany wprowadzone do bazy danych od ostatniej transakcji. Jest używany po operacjach INSERT, UPDATE lub DELETE, aby zapisać zmiany na stałe w bazie.

```
cursor = conn.cursor()
table_name = "students"
create_table_query = f"""
CREATE TABLE IF NOT EXISTS {table_name} (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT NOT NULL,
    email VARCHAR(100) NOT NULL,
```

```
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
"""
cursor.execute(create_table_query)
print(f"Tabela '{table_name}' została utworzona.")
```

```
Tabela 'students' została utworzona.
```

3.1. INSERT - dodawanie danych do tabeli

```
insert_query = f"INSERT INTO {table_name} (name, age, email) VALUES
(%s, %s, %s)"
students_data = [
    ("Anna Kowalska", 22, "anna.kowalska@example.com"),
    ("Jan Nowak", 25, "jan.nowak@example.com"),
    ("Maria Wiśniewska", 20, "maria.wisniewska@example.com")
]
cursor.executemany(insert_query, students_data)
conn.commit()
print("Dodano dane do tabeli.")
```

```
Dodano dane do tabeli.
```

3.2. SELECT - odczyt danych z tabeli

```
select_query = f"SELECT * FROM {table_name}"
cursor.execute(select_query)
print("Dane w tabeli:")
for row in cursor.fetchall():
    print(row)
```

```
Dane w tabeli:
(1, 'Anna Kowalska', 22, 'anna.kowalska@example.com',
datetime.datetime(2024, 11, 17, 15, 55, 38))
(2, 'Jan Nowak', 25, 'jan.nowak@example.com', datetime.datetime(2024,
11, 17, 15, 55, 38))
(3, 'Maria Wiśniewska', 20, 'maria.wisniewska@example.com',
datetime.datetime(2024, 11, 17, 15, 55, 38))
```

3.3. UPDATE - aktualizacja danych w tabeli

```
update_query = f"UPDATE {table_name} SET age = age + 1 WHERE name =
'Jan Nowak'"
cursor.execute(update_query)
conn.commit()
print("Zaktualizowano dane w tabeli (wiek Jana Nowaka zwiększono o
1).")
```

```
Zaktualizowano dane w tabeli (wiek Jana Nowaka zwiększono o 1).
```

3.4. DELETE - usunięcie danych z tabeli

```
delete_query = f"DELETE FROM {table_name} WHERE age > 22"
cursor.execute(delete_query)
conn.commit()
print("Usunięto dane z tabeli (osoby starsze niż 22 lata).")
```

```
Usunięto dane z tabeli (osoby starsze niż 22 lata).
```

---

4. Obsługa transakcji

Transakcje to zestaw operacji na bazie danych, które są wykonywane jako jedna logiczna jednostka pracy. Oznacza to, że wszystkie operacje wchodzące w skład transakcji muszą zakończyć się sukcesem, aby zostały zapisane w bazie danych. Jeśli którakolwiek operacja się nie powiedzie, wszystkie zmiany wprowadzone w ramach transakcji są wycofywane.

- Atomicity - atomowość
- Consistency - spójność
- Isolation - izolacja
- Durability - trwałość
- `.start_transaction()` - rozpoczyna nową transakcję
- `.commit()` - zatwierdza transakcję zapisując zmiany w bazie danych
- `.rollback()` - wycofuje transakcję przywracając bazę do stanu przed jej wywołaniem.

```
try:
    conn.start_transaction()

    insert_query = f"INSERT INTO {table_name} (name, age, email)
VALUES (%s, %s, %s)"
    cursor.execute(insert_query, ("Adam Kowalski", 30,
"adam.kowalski@example.com"))
    print("Dodano rekord: Adam Kowalski.")

    update_query = f"UPDATE {table_name} SET age = age + 1 WHERE name
= 'Adam Kowalski'"
    cursor.execute(update_query)
    print("Zaktualizowano wiek Adama Kowalskiego.")


    if True: # Symulowanie błędu
        raise Exception("Symulowany błąd podczas transakcji!")
```

```
    conn.commit()
    print("Transakcja została zatwierdzona.")

except Exception as e:
    conn.rollback()
    print(f"Błąd transakcji: {e}. Wycofano zmiany.")

cursor.execute(f"SELECT * FROM {table_name}")
print("Dane w tabeli:")
for row in cursor.fetchall():
    print(row)
```

```
Dodano rekord: Adam Kowalski.
Zaktualizowano wiek Adama Kowalskiego.
Błąd transakcji: Symulowany błąd podczas transakcji!. Wycofano zmiany.
Dane w tabeli:
(1, 'Anna Kowalska', 22, 'anna.kowalska@example.com',
datetime.datetime(2024, 11, 17, 15, 55, 38))
(3, 'Maria Wiśniewska', 20, 'maria.wisniewska@example.com',
datetime.datetime(2024, 11, 17, 15, 55, 38))
```

---

5. Zapytania dynamiczne i parametryzowane

- Dynamiczne budowanie zapytań SQL - Pisanie elastycznych zapytań zależnych od danych wejściowych.
- Zapytania parametryzowane - Zabezpieczenie przed SQL Injection.

5.1. Dynamiczne budowanie zapytań SQL

Dynamiczne zapytania SQL pozwalają na tworzenie elastycznych i dostosowywanych do potrzeb zapytań w czasie działania programu. Są szczególnie użyteczne w przypadkach, gdy struktura lub warunki zapytania zależą od zmiennych, które nie są znane w momencie pisania kodu.

Dynamiczne zapytania mogą być niebezpieczne, jeśli są źle używane:

- Jeśli dane wejściowe użytkownika są wstawiane do zapytania bez walidacji lub parametryzacji, mogą zostać wykorzystane do złośliwych działań.
- Dynamiczne zapytania mogą być trudniejsze do debugowania i utrzymania, jeśli są zbyt rozbudowane.

```
column_name = "age"
condition = "age > 20"
dynamic_query = f"SELECT {column_name} FROM {table_name} WHERE
{condition}"
print(f"Dynamicznie wygenerowane zapytanie: {dynamic_query}")

cursor.execute(dynamic_query)
dynamic_results = cursor.fetchall()
print("Wyniki zapytania dynamicznego:", dynamic_results)
```

5.2. Zapytania parametryzowane

- Chronią przed SQL Injection – jedna z najczęstszych luk bezpieczeństwa.
- Automatycznie oczyszczają dane wejściowe użytkownika.
- Poprawiają wydajność i czytelność kodu.
- Są standardem w nowoczesnych aplikacjach, które wymagają bezpiecznego dostępu do baz danych.

```
insert_query = f"INSERT INTO {table_name} (name, age, email) VALUES
(%s, %s, %s)"
student_data = ("Jan Kowalski", 25, "jan.kowalski@example.com")

cursor.execute(insert_query, student_data)
conn.commit()
print("Dodano rekord do tabeli za pomocą zapytania
parametryzowanego.")
```

```
Dodano rekord do tabeli za pomocą zapytania parametryzowanego.
```

Pobieranie wyników w bezpieczny sposób

```
select_query = f"SELECT * FROM {table_name} WHERE name = %s"
name_to_search = "Jan Kowalski"
cursor.execute(select_query, (name_to_search,))
results = cursor.fetchall()
print("Wyniki wyszukiwania dla użytkownika 'Jan Kowalski':", results)
```

```
Wyniki wyszukiwania dla użytkownika 'Jan Kowalski': [(6, 'Jan
Kowalski', 25, 'jan.kowalski@example.com', datetime.datetime(2024, 11,
17, 16, 58, 21))]
```

Zabezpieczenie przed SQL Injection

```
unsafe_name = "'; DROP TABLE students; --"
```

Niezabezpieczone zapytanie (NIE UŻYWAĆ W PRAKTYCE):

```
unsafe_query = f"SELECT * FROM {table_name} WHERE name =
'{unsafe_name}'"
print(f"Niebezpieczne zapytanie: {unsafe_query}")
```

```
Niebezpieczne zapytanie: SELECT * FROM students WHERE name = ''; DROP
TABLE students; --'
```

Parametryzowane zapytania automatycznie oczyszczają dane wejściowe użytkownika, dzięki czemu specjalne znaki (np. `'`, `;`) nie zostaną potraktowane jako część kodu SQL.

Bezpieczna wersja:

```
cursor.execute(select_query, (unsafe_name,))
safe_results = cursor.fetchall()
print("Bezpieczne wyniki (zapytanie parametryzowane):", safe_results)

Bezpieczne wyniki (zapytanie parametryzowane): []
```

---

6. Użycie procedur składowanych i funkcji

6.1. Wywołanie procedur z poziomu Pythona

```
DELIMITER //

CREATE PROCEDURE AddStudent(IN student_name VARCHAR(100), IN student_age INT, IN student_email VARCHAR(100))
BEGIN
    INSERT INTO students (name, age, email) VALUES (student_name, student_age, student_email);
END //

DELIMITER ;
```

- `.callproc()` - funkcja do wywoływania procedur

```
procedure_name = "AddStudent"

student_name = "Maria Kowalska"
student_age = 28
student_email = "maria.kowalska@example.com"

cursor.callproc(procedure_name, [student_name, student_age,
student_email])
conn.commit()
print(f"Procedura '{procedure_name}' została wywołana. Dodano
studenta: {student_name}.")

Procedura 'AddStudent' została wywołana. Dodano studenta: Maria
Kowalska.
```

6.2. Wywołanie funkcji z poziomu Pythona

```
DELIMITER //

CREATE FUNCTION GetStudentCount() RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE student_count INT;
    SELECT COUNT(*) INTO student_count FROM students;
    RETURN student_count;
END //

DELIMITER ;
```

- `.fetchone()` - funckja zwracająca pojedynczą wartość

```
cursor.execute("SELECT GetStudentCount() AS student_count")
result = cursor.fetchone()
print(f"Liczba studentów w tabeli: {result[0]}")

Liczba studentów w tabeli: 4
```

---

7. Połączenia z bazą danych w środowisku produkcyjnym

- **Pooling** – Korzystanie z puli połączeń dla wydajności.
- Obsługa błędów i stabilność połączeń.

W rzeczywistych projektach (np. aplikacje działające na serwerach) zarządzanie połączeniami jest krytyczne dla wydajności i niezawodności.

```
from mysql.connector import pooling, Error
```

7.1. Tworzenie puli połączeń

Pooling połączeń (pula połączeń) to technika zarządzania połączeniami z bazą danych, która polega na utrzymywaniu zestawu (puli) wcześniej utworzonych połączeń. Zamiast otwierać i zamykać nowe połączenie za każdym razem, aplikacja może:

1. Pobrać połączenie z puli, gdy go potrzebuje.
2. Zwrócić połączenie do puli, gdy skończy z niego korzystać.

Dzięki temu oszczędza się czas i zasoby, ponieważ tworzenie nowego połączenia do bazy danych jest operacją kosztowną.

- `pool_reset_session` - resetowanie puli przed każdym użyciem

```
connection_pool = pooling.MySQLConnectionPool(
    pool_name="mypool",
    pool_size=5,
    pool_reset_session=True,
    host="localhost",
    user="root",
    password="root",
    database="student_database"
)
print("Pula połączeń została utworzona.")
```

```
Pula połączeń została utworzona.
```

7.2. Pobieranie połączenia z puli

- `.get_connection()` - pobiera połączenie z puli
- `.is_connected()` - sprawdza czy nadal jest połączenie z serwerem

```
try:
    conn = connection_pool.get_connection()
    cursor = conn.cursor()

    query = "SELECT COUNT(*) AS total_students FROM students"
    cursor.execute(query)
    result = cursor.fetchone()
    print(f"Liczba studentów w tabeli: {result[0]}")

except Error as e:
    print(f"Błąd połączenia: {e}")

finally:
    # Zamknięcie połączenia (wraca do puli)
    if conn.is_connected():
        cursor.close()
        conn.close()
        print("Połączenie zostało zamknięte i zwrócone do puli.")
```

```
Liczba studentów w tabeli: 4
Połączenie zostało zamknięte i zwrócone do puli.
```

7.3. Obsługa błędów i stabilność połączeń

Dzięki obsłudze wyjątków aplikacja może automatycznie reagować na problemy z połączeniem, takie jak:

- Przeciążenie serwera baz danych.

- Utrata połączenia z bazą.

```
def execute_query_with_retry(query, max_retries=3):
    retries = 0
    while retries < max_retries:
        try:
            conn = connection_pool.get_connection()
            cursor = conn.cursor()
            cursor.execute(query)
            results = cursor.fetchall()
            print("Zapytanie wykonane pomyślnie:", results)
            return results
        except Error as e:
            retries += 1
            print(f"Błąd połączenia (próba {retries}/{max_retries}):
{e}")
        finally:
            if conn.is_connected():
                cursor.close()
                conn.close() # W przypadku pooling, połączenie
fizycznie nie jest zamknięte
                print("Połączenie zostało zamknięte i zwrócone do
puli.")
    print("Nie udało się wykonać zapytania po kilku próbach.")
    return None

query = "SELECT name FROM students WHERE age > 20"
execute_query_with_retry(query)
```

```
Zapytanie wykonane pomyślnie: [('Anna Kowalska',), ('Jan Kowalski',),
('Maria Kowalska',)]
Połączenie zostało zamknięte i zwrócone do puli.

[('Anna Kowalska',), ('Jan Kowalski',), ('Maria Kowalska',)]
```

Zadania

Kod do utworzenia bazy danych znajduje się w pliku `baza.sql`

1. Tworzenie modułów do zarządzania bazą

- `db_connection.py` - Moduł odpowiedzialny za zarządzanie połączeniem z bazą danych.
- `product_manager.py` - Moduł zarządzający operacjami na tabeli `products`.
- `transaction_manager.py` - Moduł zarządzający tabelą `transactions`.

---

2. Zarządzanie produktami w magazynie

W module `product_manager.py` zaimplementuj funkcje:

- `add_product(name, quantity, price)` - Dodaje nowy produkt do magazynu.
- `get_all_products()` - Zwraca wszystkie produkty w magazynie.
- `update_quantity(product_id, new_quantity)` - Aktualizuje ilość produktu na stanie.
- `delete_product(product_id)` - Usuwa produkt na podstawie jego ID.

---

3. Obsługa transakcji

W module `transaction_manager.py` zaimplementuj funkcje:

- `record_transaction(product_id, quantity_change, transaction_type)` - Zapisuje operację na produkcie do tabeli `transactions`.
- `get_all_transactions()` - Pobiera wszystkie zarejestrowane transakcje.

---

4. Główna aplikacja

1. Utwórz plik `app.py`, który połączy moduły `product_manager.py` i `transaction_manager.py`.
2. Dodaj interaktywne menu:
   - `1: Dodaj nowy produkt.`
   - `2: Wyświetl wszystkie produkty.`
   - `3: Zaktualizuj ilość produktu.`
   - `4: Usuń produkt.`
   - `5: Wyświetl historię transakcji.`
   - `6: Wyjdź.`