**SQLite3 w Pythonie**

**Wstęp**

Po poznaniu domeny językowej SQL-a (czyli m.in, jak tworzyć polecenia i z jakich struktur składa się baza danych), nadszedł czas na zapoznanie się z tym, jak integrować kod Pythona z bliblioteką SQLite3. Czyli, jak w pełni zautomatyzować proces obsługi bazy danych i wywoływania odpowiednich poleceń SQL.

Aby jednak poznać wszystkie możliwości, musimy rozbić proces integracji kodu z bazą na czynniki pierwsze. Czyli po pierwsze - jakie polecenia wykonać, aby nawiązać połączenie z bazą z poziomu Pythona, w jaki sposób reprezentowana jest baza z danymi, jak wykonywać podstawowe polecenia CRUD (Create, Read, Update, Delete) oraz jak zbudować taką funkcjonalność w podejściu obiektowym? Nie owijając dłużej w bawełnę, przejdźmy do omówienia pierwszych zagadnień ze szkolenia.

**Utworzenie bazy danych**

Baza danych, z którą będziemy nawiązywali komunikację ma postać pliku o rozszerzeniu .sqlite3. Co ciekawe, utworzenie takiego pliku możemy przeprowadzić na dwa sposoby: albo przez jego ręczne utworzenie z poziomu eksploratora lub zakodować ten proces w Pythonie (co według mnie jest o wiele bardziej efektywnym rozwiązaniem). Taki plik będzie bowiem automatycznie tworzony podczas próby nawiązania połączenia z konkretną bazą (która jeszcze nie istnieje).

**Przykład**

Jeżeli zakodujemy, że chcemy nawiązać połączenie, z bazą w postaci pliku o nazwie: db.sqlite3, system rozpocznie jego poszukiwanie. Jeżeli taki zasób zostanie odnaleziony, to nawiąże się z nim komunikacja. W przeciwnym razie - plik ten zostanie automatycznie utworzony. Dzięki takiemu rozwiązaniu programista jest zwolniony z konieczności pamiętania o tym, żeby ręcznie utworzyć własną bazę. Pełna wygoda i prostota.

Przeanalizujmy, jak to może być zrealizowane w kodzie.

Nawiązywanie połączenia z bazą:

1. W pierwszej kolejności do tworzonego Pythonowego pliku, zaimportujemy bibliotekę sqlite3.
2. Następnie, do nawiązywania połączenia z bazą, zbudujemy funkcję connect\_to\_db. Skorzystamy w niej z metody sqlite3.connect(ścieżka\_do\_db). Funkcja ta nawiązuje połączenie z podanym plikiem (bazą) oraz zwraca handler umożliwiający zarządzanie tym zasobem. Zapamiętamy go pod zmienną con, z której będziemy korzystali przy wykonywaniu operacji na bazie, np. tworzenie tabeli, dodawanie rekordu itp.

| import sqlite3  def connect\_to\_db(path):  con = sqlite3.connect(path)  return con  con = connect\_to\_db("example-database.sqlite3") |
| --- |

Przypominam Ci, że wywoływana metoda sqlite3.connect działa w następujący sposób:

* Poszukuje plik bazy danych pod podaną przez użytkownika ścieżką
* Jeżeli plik został odnaleziony, to jest z nim nawiązywane połączenie
* W przeciwnym razie, plik jest tworzony i dopiero wtedy nawiązywane jest z nim połączenie

**Utworzenie tabeli**

Po stworzeniu bazy danych, jesteśmy gotowi na wypełnianie jej odpowiednimi strukturami danych. W pierwszej kolejności zadbamy o utworzenie odpowiednich tabel. Na razie stworzę prostą tabelę, np. Customers o kilku kolumnach (id, name, surname, date\_joined).

Nie chcę zbytnio komplikować tego przykładu z myślą o tym, abyś mógł łatwo poznać wszystkie podstawowe polecenia sqlite3. Na zakończenie tego szkolenia będę chciał jednak zaimplementować bardziej złożoną relację między tabelami w bazie - taką, jaką stworzyliśmy w szkoleniu SQLite.

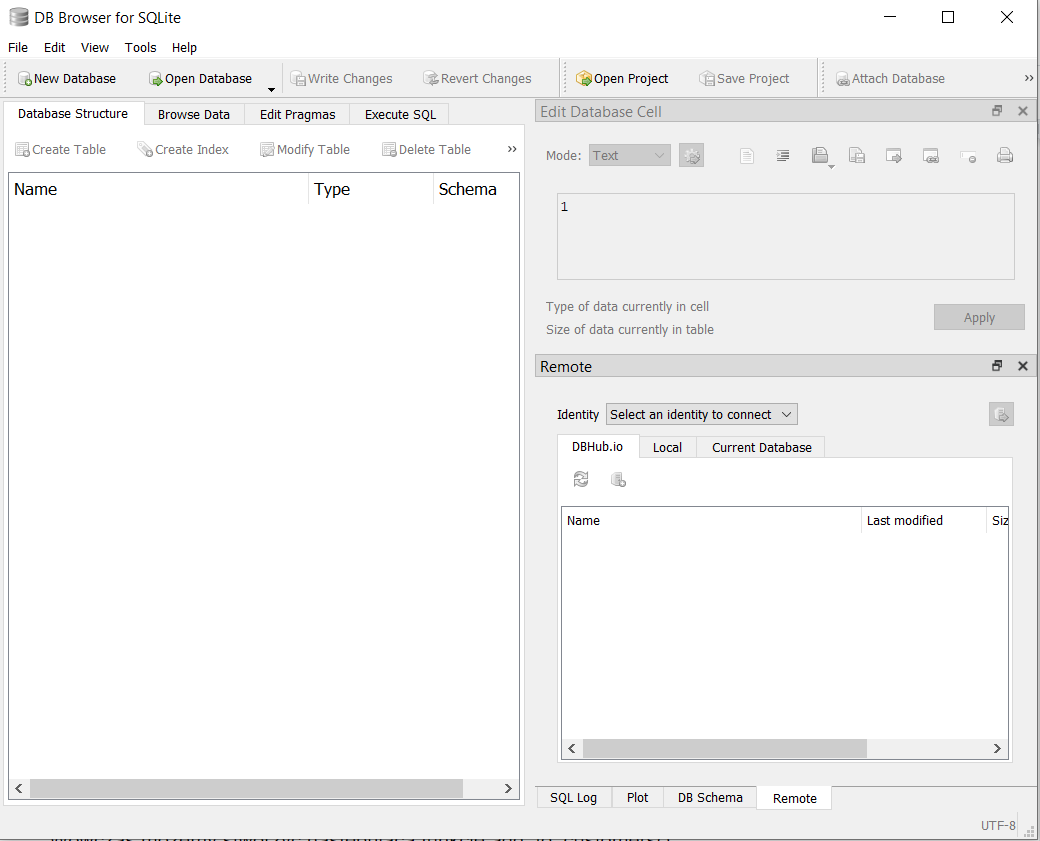
Tabelę Customers stworzymy przy użyciu funkcji create\_table().

| import sqlite3  def connect\_to\_db(path):  con = sqlite3.connect(path)  return con  def create\_table(con):  query = "CREATE TABLE IF NOT EXISTS Customers(id INTEGER PRIMARY KEY, name TEXT NOT NULL, surname TEXT NOT NULL, date\_joined DATE NOT NULL);"  con.execute(query)  con.commit()  con = connect\_to\_db("example-database.sqlite3") create\_table(con) |
| --- |

W funkcji tej umieściliśmy zapytanie, jakie chcemy wykonać CREATE TABLE IF NOT EXISTS Customers(id INTEGER PRIMARY KEY, name TEXT NOT NULL, surname TEXT NOT NULL, date\_joned DATE NOT NULL); i polecenie to wykonaliśmy dzięki metodom execute() oraz commit().   
  
Pierwsza metoda służy bezpośrednio do wykonania polecenia, a commit() do zatwierdzenia zmian. Pamiętaj szczególnie o wywoływaniu wspomnianego polecenia commit() - tak naprawdę konieczne będzie odwoływanie się do niego przy każdym zapytaniu z rodziny DML. Zwróć również uwagę na konieczność wysyłania handlera (con) połączenia, który utworzyliśmy i zwróciliśmy z funkcji connect\_to\_db.

**UWAGA:**

Po każdej operacji modyfikującej zawartość bazy danych, warto by było sprawdzać, czy faktycznie baza została zmodyfikowana. Niestety pliku z rozszerzeniem .sqlite3 nie otworzysz za pomocą standardowego edytora tekstu. Dlatego, aby móc wygodnie nadzorować manipulacje na bazie, zainstalujemy aplikację [DB Browser for SQLite](https://sqlitebrowser.org/).



*Widok na uruchomioną aplikację DB Browser*

**Dodawanie rekordów do tabeli**

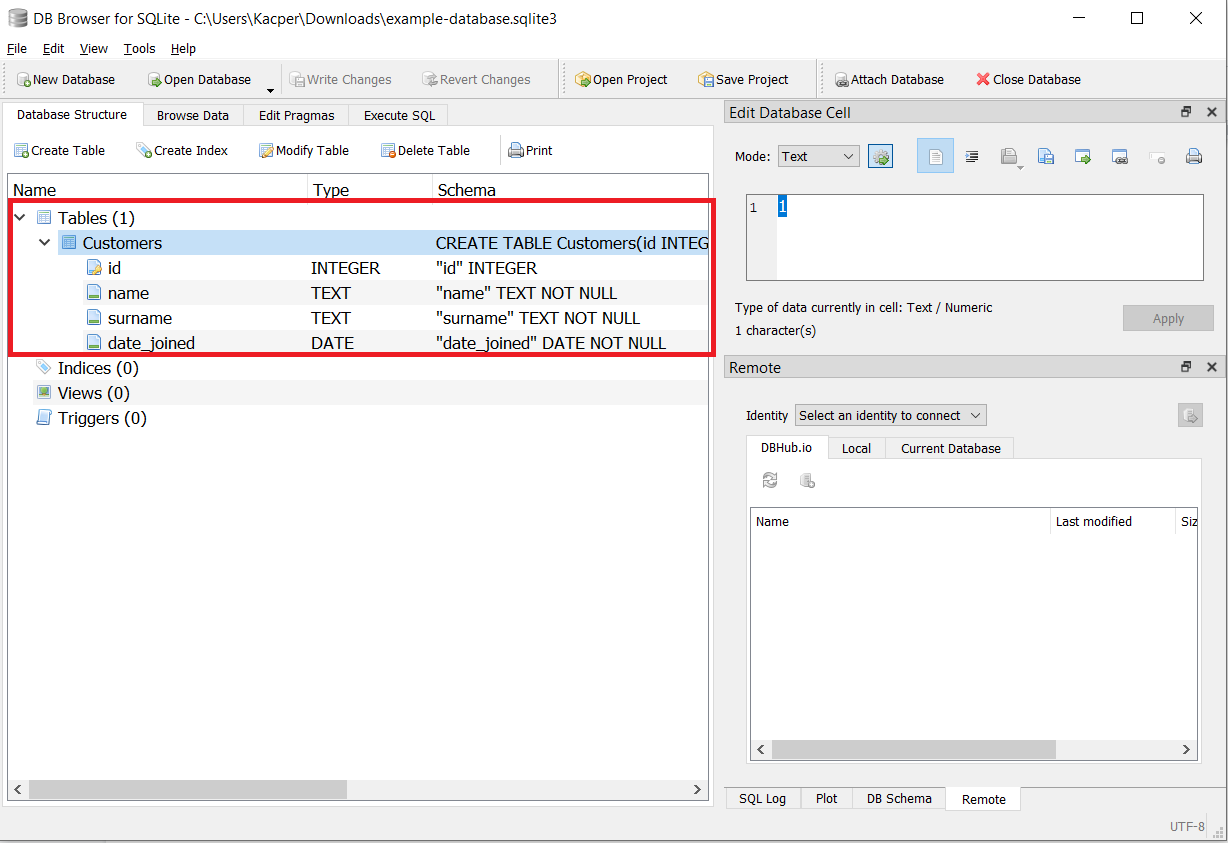
W porządku! Baza danych utworzona, pierwszą tabelę również już mamy. Nadeszła teraz pora na dodanie do tabeli pierwszych rekordów. Załóżmy, że będziemy chcieli utworzyć pierwszego Customer-a o poniższych danych:

* Name: John
* Surname: Wick
* Date\_joined: 2000-09-02

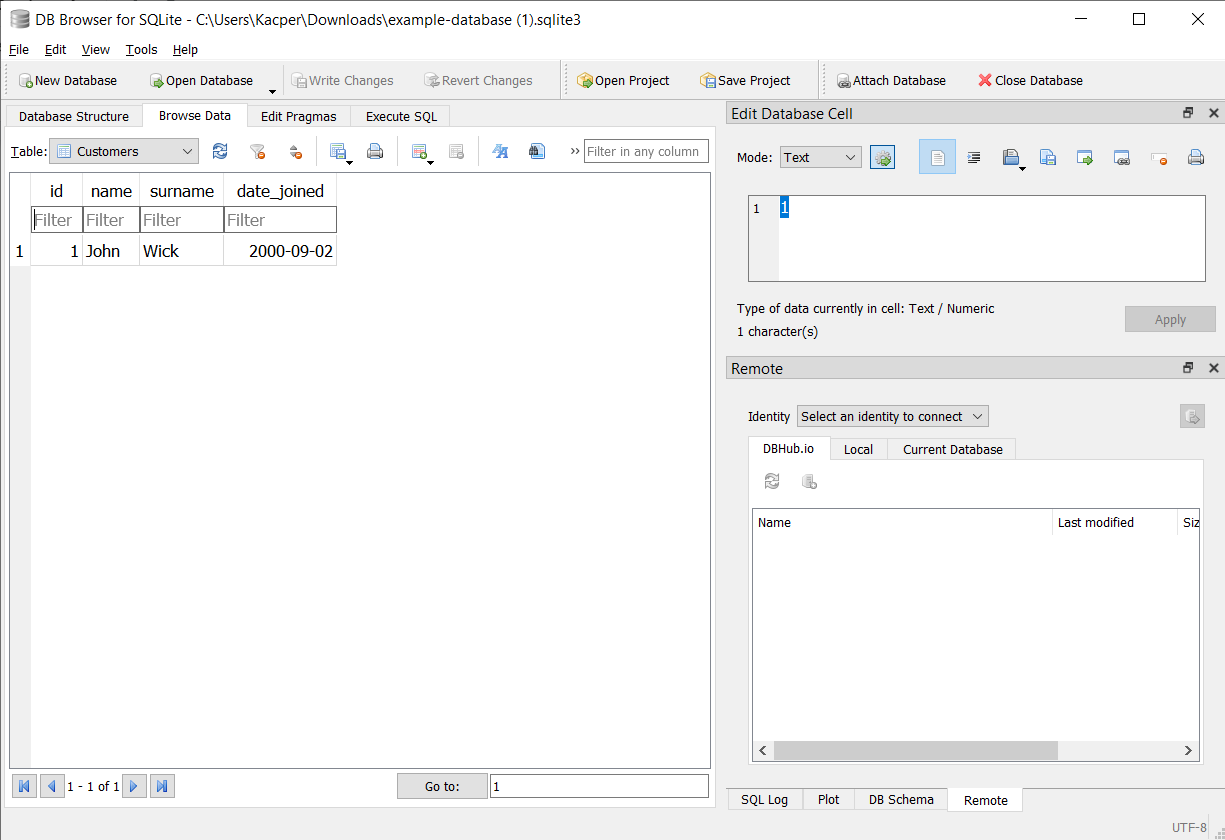
Wówczas możemy stworzyć następującą funkcję add\_to\_customers(conn):

| import sqlite3  def connect\_to\_db(path):  con = sqlite3.connect(path)  return con  def create\_table(con):  query = "CREATE TABLE IF NOT EXISTS Customers(id INTEGER PRIMARY KEY, name TEXT NOT NULL, surname TEXT NOT NULL, date\_joined DATE NOT NULL);"  con.execute(query)  con.commit()  def add\_to\_customers(con):  query = "INSERT INTO Customers(name, surname, date\_joined) VALUES('John', 'Wick', '2000-09-02')"  con.execute(query)  con.commit()  con = connect\_to\_db("example-database.sqlite3") create\_table(con) add\_to\_customers(con) |
| --- |

Po zbudowaniu takiego programu i jego uruchomieniu, zostanie dodany nowy klient do tabeli. Chcąc upewnić się, czy wszystko przebiegło zgodnie z oczekiwaniami, podejrzyjmy zawartość bazy example-database.sqlite3 z poziomu aplikacji DB Browser for SQLite.



*Utworzone tabele w bazie*



*Dane w tabeli Customers*

**Ulepszenie funkcji…**

Super! Udało nam się dodać pierwszy rekord do tabeli… Jednak przyznam szczerze, że takie rozwiązanie na dodawanie rekordów do bazy jest niezbyt efektywne. A to dlatego, że w poleceniu INSERT na sztywno podaliśmy wartości danych kolumn, które mają zostać umieszczone w Customers. Lepiej by było stworzyć funkcję add\_customer która nie będzie zawsze dodawała tego samego klienta (John’a Wick’a), tylko tak naprawdę dowolnego klienta, dla którego prześlemy odpowiednie wartości name, surname oraz date\_joined.   
  
Aby tego dokonać, zmodyfikujemy nieco omawianą funkcję do takiej postaci:

| def add\_to\_customers(con, name, surname, date\_joined):  query = "INSERT INTO Customers(name, surname, date\_joined) VALUES(?, ?, ?)"  con.execute(query, (name, surname, date\_joined))  con.commit() |
| --- |

A jej wywołanie mogłoby prezentować się następująco:

| add\_to\_customers(con, 'John', 'Wick', '2000-09-02') add\_to\_customers(con, 'James', 'Bond', '2002-05-16') |
| --- |

Dzięki takiemu rozwiązaniu, możesz łatwo dodawać nowych klientów do bazy i w pełni dostosowywać wartości pól. Niezwykle wygodne rozwiązanie!

Trzeba jednak pamiętać o odpowiednim zapisie tworzonych poleceń...

**Czym są ? podane w INSERT?**

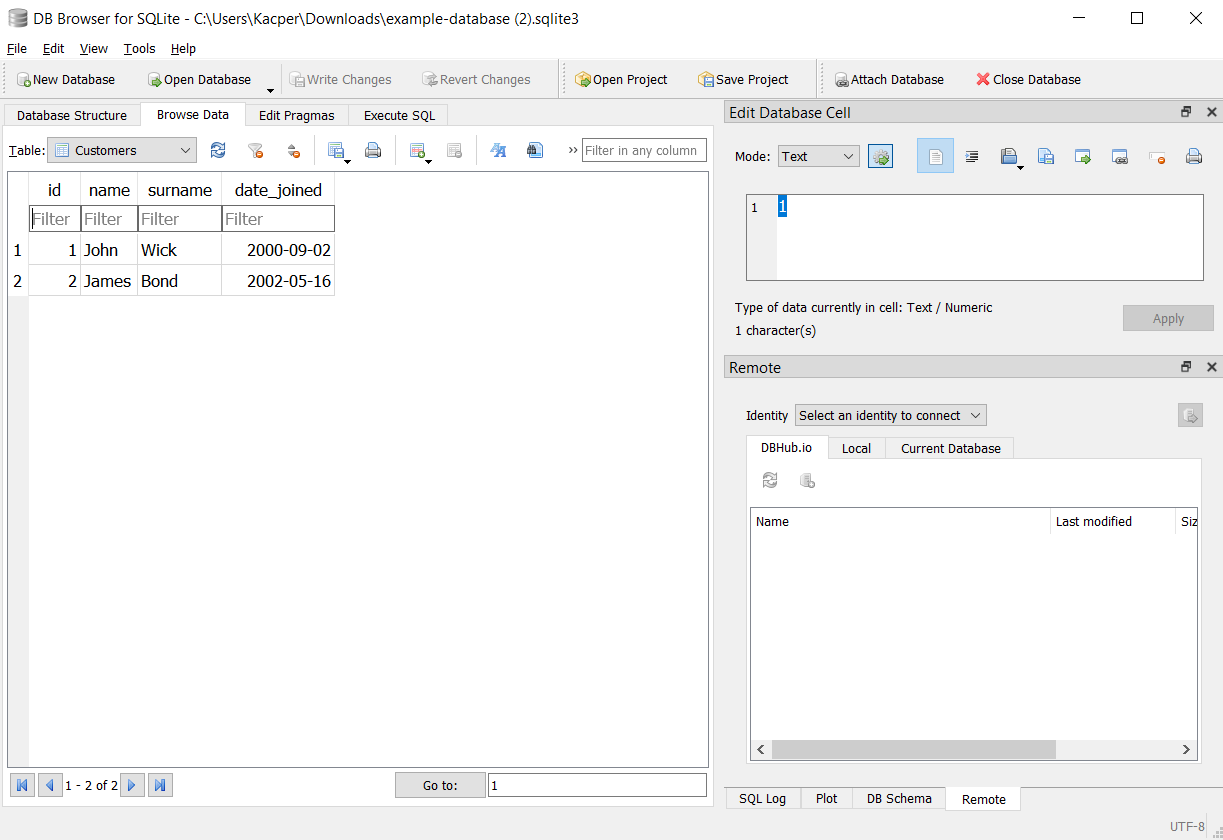
Otóż zauważ, że w wyniku powyższej zmiany, znacznej modyfikacji uległo zapytanie query, które obecnie wygląda następująco - INSERT INTO Customers(name, surname, date\_joined) VALUES(?, ?, ?).   
  
Znalazły się w niej dość tajemnicze znaki zapytania. Umożliwiają one przygotowanie zapytania pod dowolną modyfikację - czyli wstawianie dowolnych wartości do pól. Pole te określamy w zapytaniu execute(), np. con.execute(query, (name, surname, date\_joined)). Zauważ, że zmienne te przesyłamy w postaci tupli jako drugi argument con.execute() i odpowiednio wartość name zostanie wstawiona za pierwszy znak zapytania w query, surname za drugi itd.

Ostateczna postać kodu:

| import sqlite3  def connect\_to\_db(path):  con = sqlite3.connect(path)  return con  def create\_table(con):  query = "CREATE TABLE IF NOT EXISTS Customers(id INTEGER PRIMARY KEY, name TEXT NOT NULL, surname TEXT NOT NULL, date\_joined DATE NOT NULL);"  con.execute(query)  con.commit()  def add\_to\_customers(con, name, surname, date\_joined):  query = "INSERT INTO Customers(name, surname, date\_joined) VALUES(?, ?, ?)"  con.execute(query, (name, surname, date\_joined))  con.commit()  con = connect\_to\_db("example-database.sqlite3") create\_table(con) add\_to\_customers(con, 'John', 'Wick', '2000-09-02') add\_to\_customers(con, 'James', 'Bond', '2002-05-16') |
| --- |

**Efekt**

Wyczyściłem naszą bazę danych (co możesz również przeprowadzić przez aplikację DB Browser) i wykonałem wyżej umieszczony kod. W efekcie do tabeli Customers zostały dodane następujące rekordy:



*Tabela Customers z dwoma rekordami*

**SELECT w kodzie**

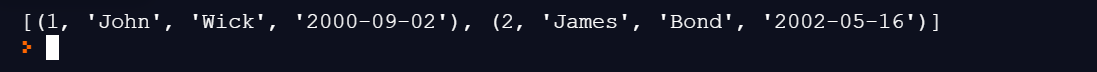
Następnym zagadnieniem, który będziemy musieli poznać jest wykonywanie polecenia SELECT i odczytywanie zwróconych rekordów w kodzie. Funkcja umożliwiająca przeprowadzenie takiej operacji może wyglądać następująco:

| def preview\_table(con, table\_name):  query = f"SELECT \* FROM {table\_name}"  results = con.execute(query).fetchall()  print(results) |
| --- |

Na szczególną uwagę zasługuje metoda fetchall(). Wywołujemy ją tuż po wykonaniu polecenia SQL-owego i jej zadaniem jest zwrócenie wszystkich wybranych przez SELECT-a rekordów w postaci listy. Podobnym poleceniem do fetchall() jest również fetchone(), który umożliwia pobranie tylko pojedynczego rekordu. Wywołanie tej funkcji w poniższy sposób:

| preview\_table(con, 'Customers') |
| --- |

Zaskutkuje wyświetleniem się następujących informacji:



**Kiedy f-string, a kiedy znak “?”**

Patrząc na powyższą funkcję preview\_table i zmienną query, możesz zastanawiać się, dlaczego nie użyliśmy tutaj, tzw. *parameter substitution* (znaków zapytania tak jak przy poleceniu INSERT)*,* tylko do parametryzacji zapytania użyliśmy f-string?   
  
Otóż sposób “ze znakami zapytania” sprawdza się w przypadkach, gdy chcemy dostosowywać zmienne odnoszące się **wartości danych kolumn.** W momencie, gdy potrzebujemy dostosować nie zawartość, ale strukturę np. nazwę kolumny, musimy skorzystać już ze standardowego podejścia - dostosowywanie nazw przez f-stringi.   
  
Bądź jednak uważny i nie nadużywaj tego sposobu, ponieważ z takim rozwiązaniem związane jest duże ryzyko wystąpienia podatności SQL Injection w tworzonej aplikacji. Więcej na temat opisywanego ryzyka znajdziesz tutaj: [SQL Injection Based on](https://www.w3schools.com/sql/sql_injection.asp).

**Pozostałe polecenia**

Myślę, że po poznaniu zasad pracy z powyższymi poleceniami, np. INSERT oraz SELECT, jesteś w stanie efektywnie budować polecenia z wykorzystaniem dowolnego innego query. Przetestuj na stworzonej dotąd tabeli takie operacje jak usuwanie - DELETE, update-owanie - UPDATE oraz polecenie ALTER.

My tymczasem przejdziemy do dalszego zagadnienia - czyli wykorzystywanie poleceń w programowaniu obiektowym praktyce.

**Ładniejszy sposób na owrapowanie poleceń?**

Poznaliśmy już podstawowe polecenia i system pracy, który możemy wykorzystywać, gdy chcemy operować na bazie danych w Pythonie. Zastanówmy się teraz, jak umieścić wszystkie poznane funkcje w klasie? Tak, aby zachować enkapsulację, a kod był dużo bardziej czysty.

Zrefaktoryzujmy poniższy kod:

| import sqlite3  def connect\_to\_db(path):  con = sqlite3.connect(path)  return con  def create\_table(con):  query = "CREATE TABLE IF NOT EXISTS Customers(id INTEGER PRIMARY KEY, name TEXT NOT NULL, surname TEXT NOT NULL, date\_joined DATE NOT NULL);"  con.execute(query)  con.commit()  def add\_to\_customers(con, name, surname, date\_joined):  query = "INSERT INTO Customers(name, surname, date\_joined) VALUES(?, ?, ?)"  con.execute(query, (name, surname, date\_joined))  con.commit()  def preview\_table(con, table\_name):  query = f"SELECT \* FROM {table\_name}"  results = con.execute(query).fetchall()  print(results)  con = connect\_to\_db("example-database.sqlite3") create\_table(con) add\_to\_customers(con, 'John', 'Wick', '2000-09-02') add\_to\_customers(con, 'James', 'Bond', '2002-05-16') preview\_table(con, 'Customers') |
| --- |

Na poniższą postać...

| import sqlite3  class Database:  def \_\_init\_\_(self, path):  self.con = sqlite3.connect(path)   def create\_table(self):  query = "CREATE TABLE IF NOT EXISTS Customers(id INTEGER PRIMARY KEY, name TEXT NOT NULL, surname TEXT NOT NULL, date\_joined DATE NOT NULL);"  self.con.execute(query)   def add\_to\_customers(self, name, surname, date\_joined):  query = "INSERT INTO Customers(name, surname, date\_joined) VALUES(?, ?, ?)"  self.con.execute(query, (name, surname, date\_joined))   def preview\_table(self, table\_name):  query = f"SELECT \* FROM {table\_name}"  results = self.con.execute(query).fetchall()  print(results)   def \_\_enter\_\_(self):  return self   def \_\_exit\_\_(self, ext\_type, exc\_value, traceback):  if isinstance(exc\_value, Exception):  self.con.rollback()  else:  self.con.commit()    self.con.close()   with Database('example2-data') as db:  db.create\_table()  db.add\_to\_customers('John', 'Wick', '2000-09-02')  db.add\_to\_customers('James', 'Bond', '2002-05-16')  db.preview\_table('Customers') |
| --- |

Jednym słowem cały kod proceduralny (czyli oparty na funkcjach) zamieniliśmy na obiektowy. Wykorzystywane funkcje umieściliśmy w klasie Database, z której to korzystamy, gdy chcemy przeprowadzić operacje. Zauważ jednak, że od tej pory metody manipulujące zapytaniami wywołujemy wewnątrz bloku with … as …, konieczne również było umieszczenie metod specjalnych \_\_enter\_\_ oraz \_\_exit\_\_ wewnątrz samej klasy. Pochylmy się na chwilę nad takim podejściem i pozwól, że Ci wszystko wytłumaczę co i jak.

**Klasa jako Context Manager?**

Pamiętasz szkolenie związane z pracą z plikami? Wtedy właśnie, aby skutecznie odczytywać/zapisywać zawartość z/do pliku, wykorzystywaliśmy strukturę with open(nazwa\_pliku, tryb\_pracy) as file.

Operacja ta była możliwa, ponieważ klasa odpowiedzialna za reprezentację plików zawierała, tzw. protokoły Context Managera, czyli metody specjalne \_\_enter\_\_ oraz \_\_exit\_\_. Przypomnę Ci tylko, że metody te nazywane są specjalnymi, choćby dlatego, że wywoływane są one niejawnie przy niektórych operacjach. I tak \_\_enter\_\_ jest wywoływane zawsze po napotkaniu struktury with klasa\_zawierająca\_tę\_metodę, a \_\_exit\_\_, gdy kończy się ciało wyrażenia with … as ….

I stąd właśnie, jeżeli chcemy “otwierać połączenie” z bazą danych reprezentowaną przez klasę i wywoływać na niej różne polecenia, musimy zdefiniować metody \_\_enter\_\_ (która będzie jedynie zwracała obiekt klasy) i \_\_exit\_\_ zatwierdzający wywołane polecenia i zamykający polecenie z bazą.

Zwróć również uwagę na zawarte wewnątrz metody \_\_exit\_\_ wyrażenie:

| if isinstance(exc\_value, Exception):  self.con.rollback() else:  self.con.commit() |
| --- |

Umożliwia ono rozróżnienie, czy na etapie egzekucji SQL-owych poleceń, nastąpił jakiś błąd. Jeżeli tak (wówczas warunek isinstance(exc\_value, Exception) jest True), następuje rollback wszystkich poleceń (wracamy do stanu przed ich wywołaniem) lub jeżeli nie nastąpił żaden błąd, to wykonywany jest commit, czyli ostateczne zatwierdzenie i wykonanie wszystkich operacji.

**Zalety takiego rozwiązania?**

Zastanawiasz się pewnie, po co ulepszać swój i uciekać do rozwiązania przedstawionego wyżej. Otóż wprowadzenie klasowego podejścia do obsługi bazy może wiązać się z poniższymi zaletami:

1. Większa kontrola nad połączeniem

Zauważ, że przy budowaniu i wychodzeniu z każdej kolejnej struktury with … as ..., automatycznie wywoływana jest metoda \_\_exit\_\_, a w niej zamykanie połączenia self.con.close(). Dzięki temu mamy pewność, że nigdy nie spowodujemy przeładowania programu, otwierając zbyt wiele nowych połączeń z kolejnymi bazami.

1. Możliwość enkapsulowania zmiennych

Jeżeli tylko zażyczymy sobie, aby utworzony handler połączenia, był niewidoczny dla komponentów poza klasą Database, wystarczy zrobić z niego pole prywatne. Dzięki temu zabezpieczymy się przed dostępem do niego z niepowołanego miejsca.

1. Sprytne i wygodne zastosowanie kontekstu with … as …

Dzięki użyciu takiego bloku, kod staje się o wiele bardziej przejrzysty i czysty. Widzisz dokładnie miejsca w kodzie, w których korzystasz z bazy.

**O czym pamiętać?**

Pamiętaj jednak, że opierając się na takim rozwiązaniu, gdy wywołamy kilka zapytań w obrębie jednego bloku with… open, silnik SQL będzie traktował je jako transakcje, czyli nieodłączne zapytania. Wówczas - jeżeli wyjątkiem zakończy się choćby jedno z nich, to w rezultacie nie wykona się żadne.

**Podsumowanie**

Reasumując, w szkoleniu tym przedstawiłem Ci, jak efektywnie tworzyć zapytania SQL i operować na bazie z poziomu Pythona. Jest to rozwiązanie dość generyczne i takie manipulację sprawdzą się z wykorzystaniem zarówno czystego języka jak i jego frameworków, np. Flask. Bądź jednak świadomy tego, że w projektach często wykorzystuję się, tzw. ORM-owe (Object Relational Mapping; o którym więcej przy szkoleniu z Flaska) podejście do pisania programów lub jeżeli zostajemy przy standardowym rozwiązaniu, to wprowadza się query builder-y. Dzięki temu nie musisz ręcznie pisać kodu SQL (do którego łatwo mogą się wkraść trudne do wychwycenia literówki i błędy) - tool automatycznie generuje za Ciebie zapytania. Przykładem takiego narzędzia może być: <https://github.com/kayak/pypika>.