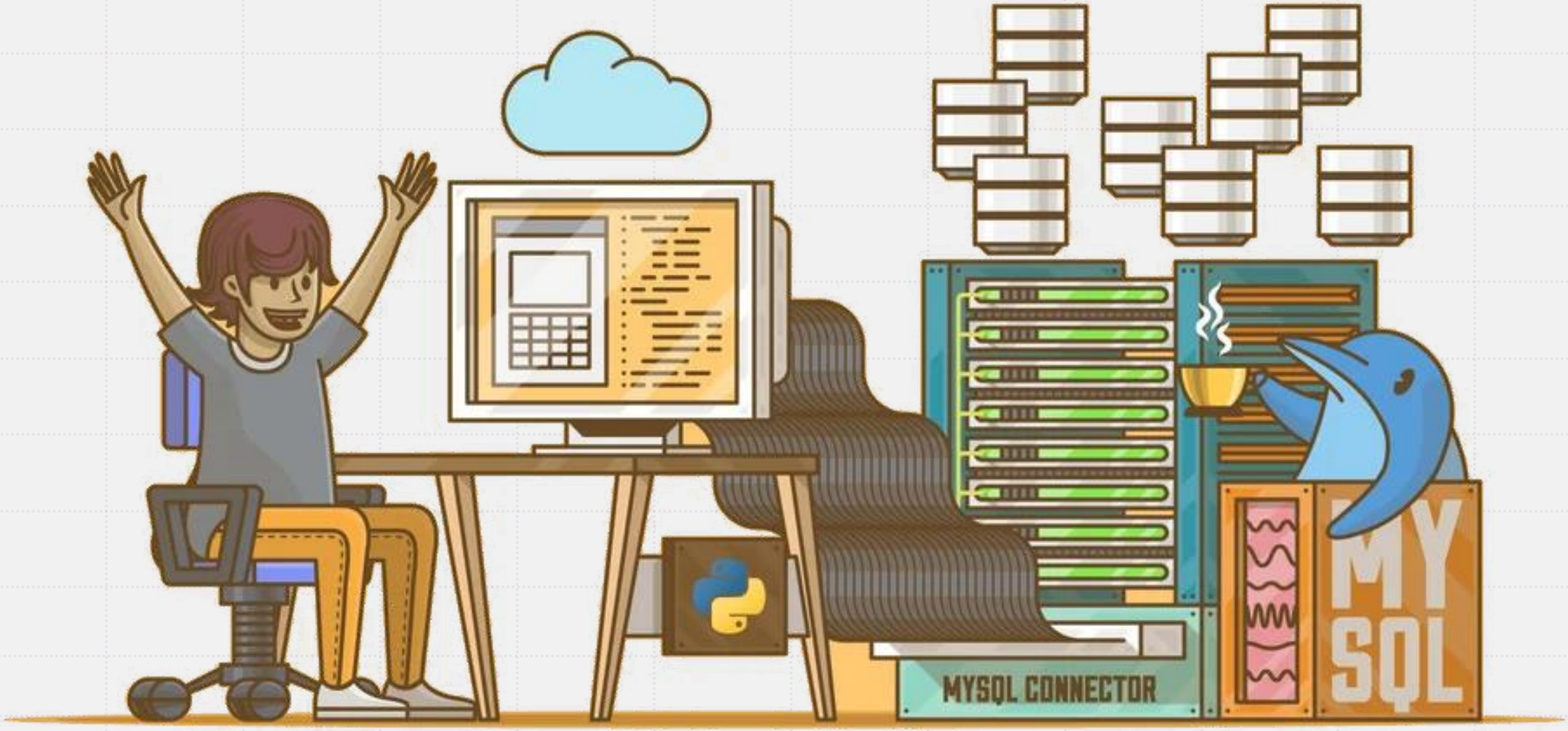
Python

Programowanie baz danych

mysql-connector-python



API

Application Programming Interface

Programistyczny interfejs aplikacji

(API - Application Programming Interface)

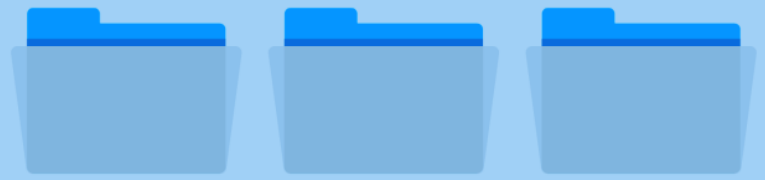
Piszemy aplikacje do komunikacji z jakimś systemem zarządzania bazą danych (np. z MySQL). Albo zastanówmy się w jaki sposób aplikacje klienckie (np. MySQL Workbench) komunikują się z aplikacją serwerową (MySQL Serwer).

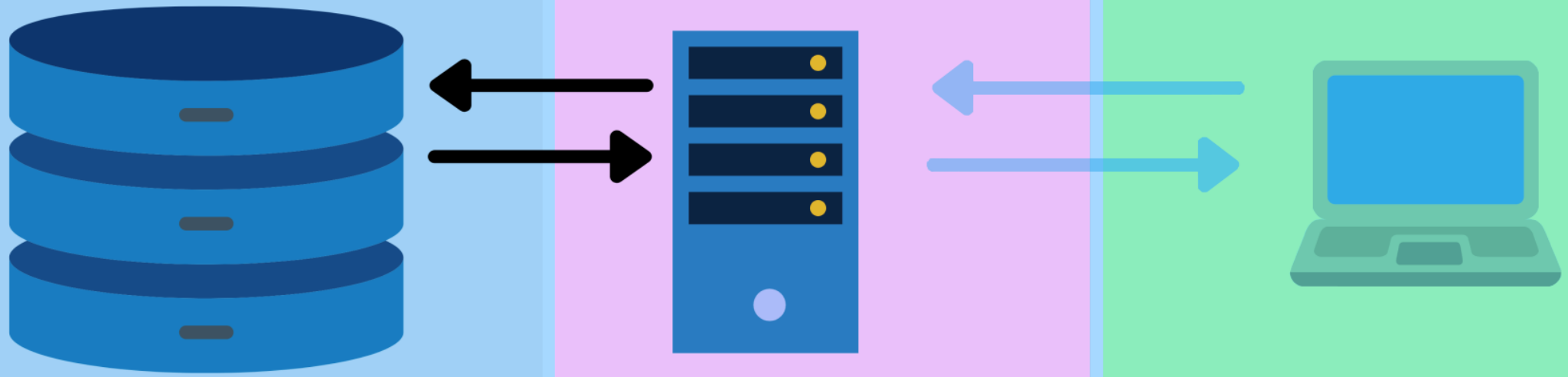
Teoretycznie każdy może stworzyć własny moduł do komunikacji z wybraną bazą danych, ze swoim zestawem funkcji/struktur i swoim formatem danych wejściowych/wyjściowych. A potem drugi moduł do komunikacji z innym DBMS, być może z innym zestawem funkcji, struktur i formatów. I początkowo tak to się właśnie wyglądało. Żeby zapanować na powstającym chaosem w latach 90 rozpoczęto pracę nad ustandaryzowaniem tego mechanizmu.

DB

DBMS

Aplikacje użytkowe

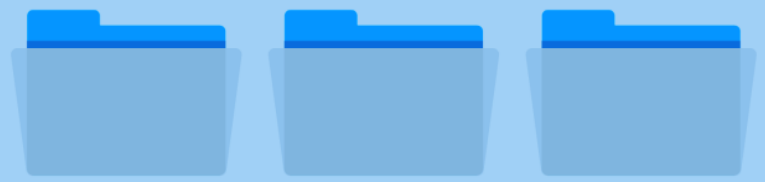


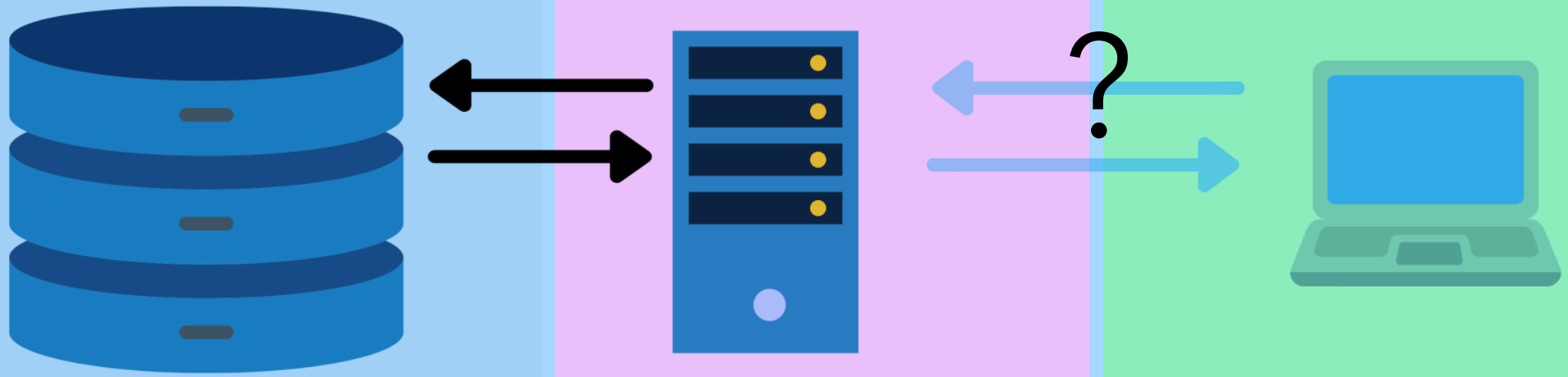


DB

DBMS

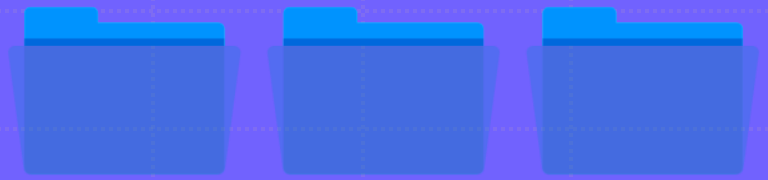
Aplikacje użytkowe



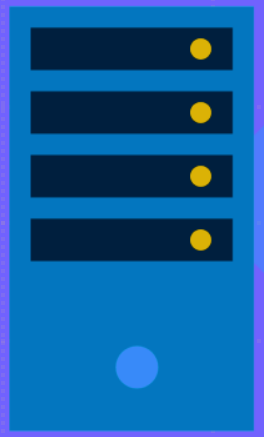


W jaki sposób komunikują się ze sobą DBMS i aplikacje użytkowe ?

Database Files

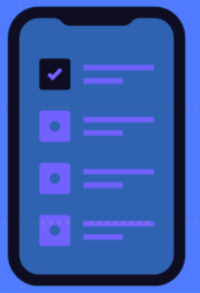


RDBMS Server



?

User Application



User Application

User Application

Programistyczny interfejs aplikacji

(API - Application Programming Interface)

W ogólności API to zestaw zasad używanych przez aplikacje do komunikacji z innymi systemami informatycznymi takimi jak system operacyjny, baza danych lub inne programy.

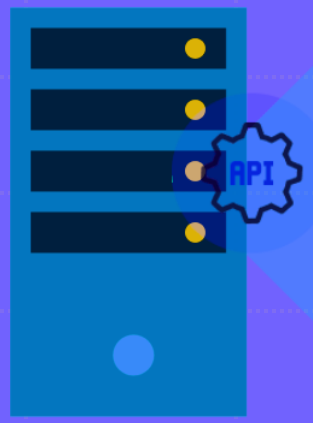
API definiuje zestaw funkcji dostępowych, które mogą być wywoływane przez aplikacje klienckie oraz format danych wejściowych i wyjściowych. API posiada program, z którym łączy się aplikacja kliencka i aplikacja kliencka musi dostosować się do udostępnionego przez ten program API.

Dzięki API programy mogą współpracować ze sobą, wymieniać informacje i wykorzystywać usługi udostępniane przez inne systemy w **ustandaryzowany sposób**.

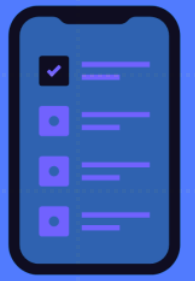
My skupimy się wyłącznie na API do programów klasy DBMS. Projektantami API mogą być zarówno sami twórcy DBMS jaki i firmy niepowiązane z DBMS oraz społeczeństwo wolnego oprogramowania. Jeżeli jakiś DBMS ma wspierać zaprojektowane API ktoś (twórcy DBMS, jakaś firma, społeczeństwo wolnego oprogramowania) musi zaimplementować moduł komunikacyjny zgodny z zaprojektowanym standardem.

Database Files

RDBMS Server

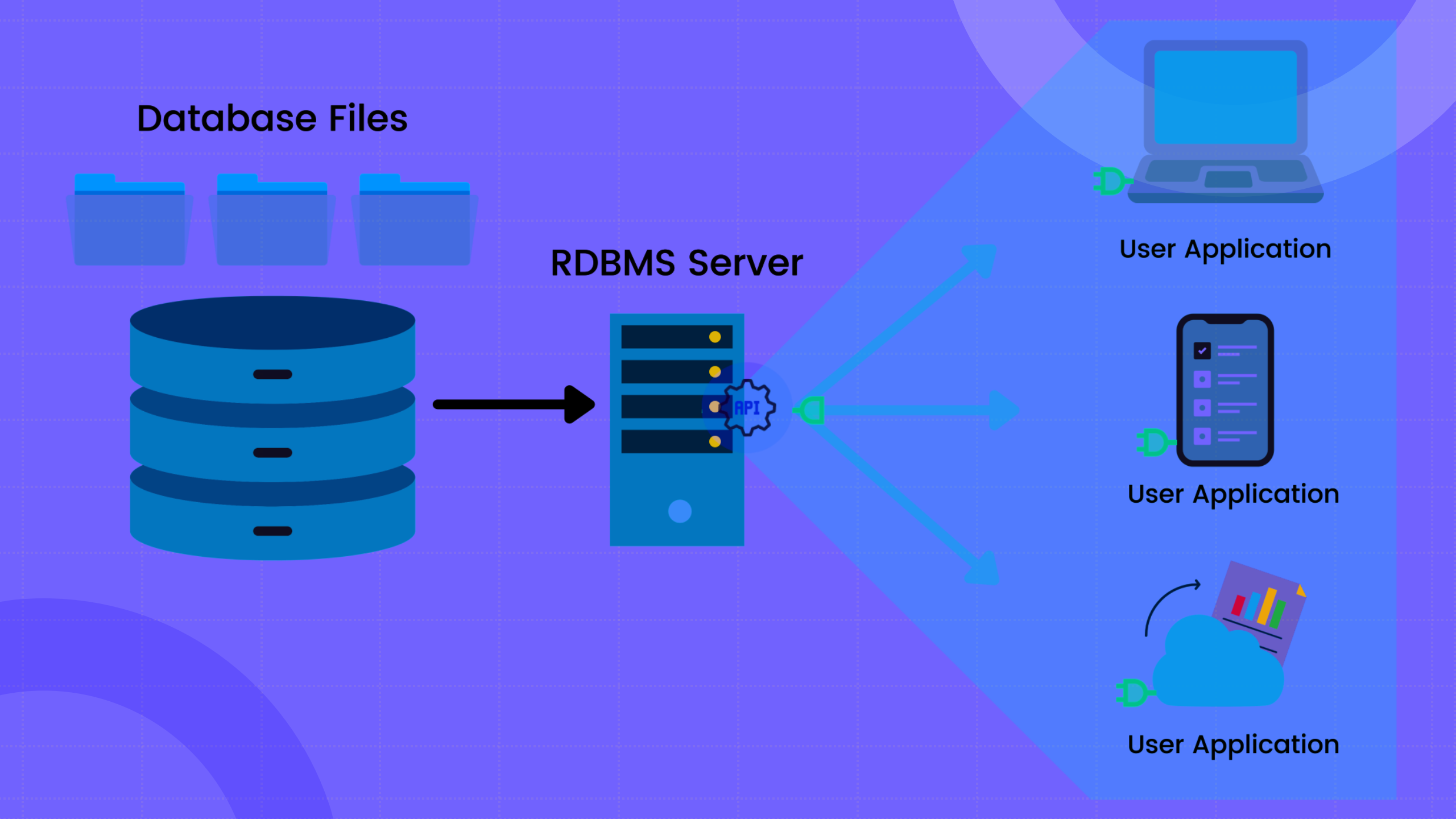


User Application



User Application

User Application


Database Files
RDBMS Server
API
User Application
User Application
User Application

ODBC i JDBC

Jednym z najpopularniejszych standardów API do DBMS jest ODBC (**O**pen **D**ata**b**ase **C**onnectivity) - opracowany przez firmę Microsoft w 1992 roku standard ogólnego przeznaczenia niezależny od platformy (windows, unix, …) oraz niepowiązany z żadnym konkretnym językiem programowania (C, Java, Python, …) ani z żadnym konkretnym DMBS (MySQL, PostgreSQL, SQLite). Większość współczesnych DBMS zawiera implementację tego standardu. Najnowsza wersja ODBC 4.0 została opracowana w 2016 roku.

Pełną specyfikację ODBC 4.0 można znaleźć na:

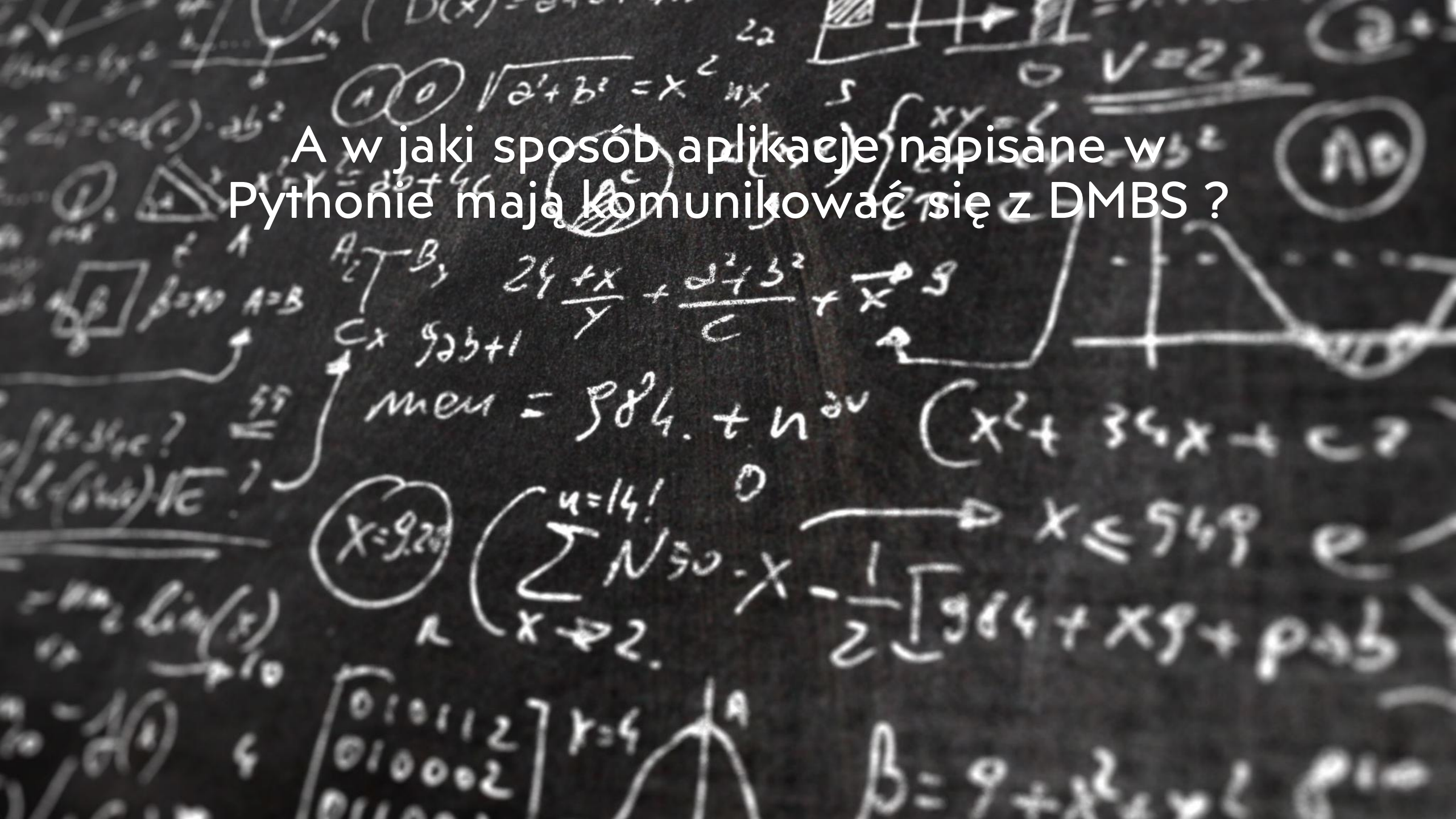
https://github.com/microsoft/ODBC-Specification/blob/master/ODBC%204.0.md

https://learn.microsoft.com/en-us/sql/odbc/reference/introduction-to-odbc?view=sql-server-ver16

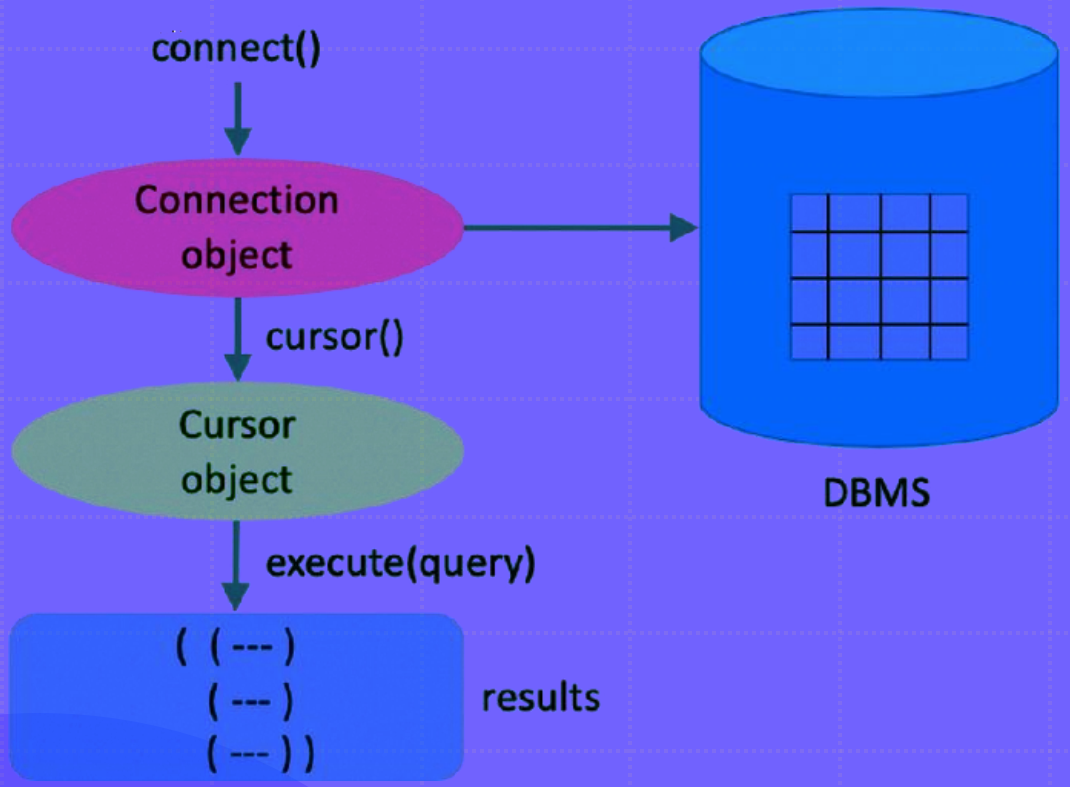
W 1997 w ślad za Microsoft fitma Sun Microsystems opracowała analogiczny standard API dla języka Java – JDBC (**J**ava **D**ata**b**ase **C**onnectivity). Najnowsza wersja JDBC 4.3 została opracowana w 2017 roku

Więcej informacji na temat JDBC 4.3 można znaleźć na:

https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/



A w jaki sposób aplikacje napisane w Pythonie mają komunikować się z DMBS ?


connect()
Connection object
cursor()
Cursor object
execute(query)
((---)
(---)
(---))
results
DBMS

DB-API

DB-API v1.0

W 1996 roku w ramach **PEP 248** została zaproponowana pierwsza wersja standardu dla modułów pythona łączących się z bazą danych, tzw. **DB-API v1.0**

Zgodnie z PEP 248 połączenie z bazą danych powinno być reprezentowane poprzez obiekt klasy Connection posiadający metody takie jak: close(), commit(), rollback(), cursor(). Metoda cursor() ma zwracać obiekt klasy Cursor reprezentujący kursor bazodanowy. Obiekty klasy Cursor powinny mieć m.in. metody: close(), execute(operation, [,params]), fetchone(), fetchmany([size]), fetchall(), …

Pełną dokumentację DB-API można znaleźć na: https://peps.python.org/pep-0248/

Contents

Page Source (GitHub)

PEP 248 – Python Database API Specification v1.0

Author: Greg Stein <gstein at lyra.org>, Marc-André Lemburg <mal at lemburg.com>
Discussions-To: Db-SIG list
Status: Final
Type: Informational
Created: 08-May-1996
Post-History:
Superseded-By: 249

▶ Table of Contents

Introduction

This API has been defined to encourage similarity between the Python modules that are used to access databases. By doing this, we hope to achieve a consistency leading to more easily understood modules, code that is generally more portable across databases, and a broader reach of database connectivity from Python.

This interface specification consists of several items:

- Module Interface
- Connection Objects
- Cursor Objects
- DBI Helper Objects

Comments and questions about this specification may be directed to the SIG on Tabular Databases in Python (http://www.python.org/sigs/db-sig).

This specification document was last updated on: April 9, 1996. It will be l
specification.

Module Interfa

https://peps.python.org/pep-0248/

DB-API v2.0

W 1999 roku, w ramach PEP 249 została zaproponowana nowa wersja DB-API obowiązująca do dziś. Nowa wersja złamała wsteczną kompatybilność. Współcześnie wszystkie moduły Pythona służące do łączenia się z oraz pracą z bazami danych implementują standard DB-API v2.0.

Pełną specyfikację DB-API v2.0 można znaleźć na: https://peps.python.org/pep-0249/

W 2001 roku Stuart Bishop otworzył dyskusję o DB-API v3.0. Z abstraktem jego propozycji można zapoznać się na https://mail.python.org/pipermail/db-sig/2001-August/001877.html

Contents

Page Source (GitHub)

PEP 249 – Python Database API Specification v2.0

Author: Marc-André Lemburg <mal at lemburg.com>
Discussions-To: Db-SIG list
Status: Final
Type: Informational
Created: 12-Apr-1999
Post-History:
Replaces: 248

▶ **Table of Contents**

Introduction

This API has been defined to encourage similarity between the Python modules that are used to access databases. By doing this, we hope to achieve a consistency leading to more easily understood modules, code that is generally more portable across databases, and a broader reach of database connectivity from Python.

Comments and questions about this specification may be directed to the SIG for Database Interfacing with Python.

For more information on database interfacing with Python and available packages see the Database Topic Guide.

This document describes the Python Database API Specification 2.0 and a set of common optional extensions. The previous version 1.0 version is still available as reference, in PEP 248. Package writers are encouraged to use this version of the specification as basis for new interfaces.

Module Interface

Constructors

https://peps.python.org/pep-0249/

Contents

Page Source (GitHub)

PEP 248 – Python Database API Specification v1.0

Author: Greg Stein <gstein at lyra.org>, Marc-André Lemburg <mal at lemburg.com>
Discussions-To: Db-SIG list
Status: Final
Type: Informational
Created: 08-May-1996
Post-History:
Superseded-By: 249

▶ **Table of Contents**

Introduction

This API has been defined to encourage similarity between the Python modules that are used to access databases. By doing this, we hope to achieve a consistency leading to more easily understood modules, code that is generally more portable across databases, and a broader reach of database connectivity from Python.

This interface specification consists of several items:

- Module Interface
- Connection Objects
- Cursor Objects
- DBI Helper Objects

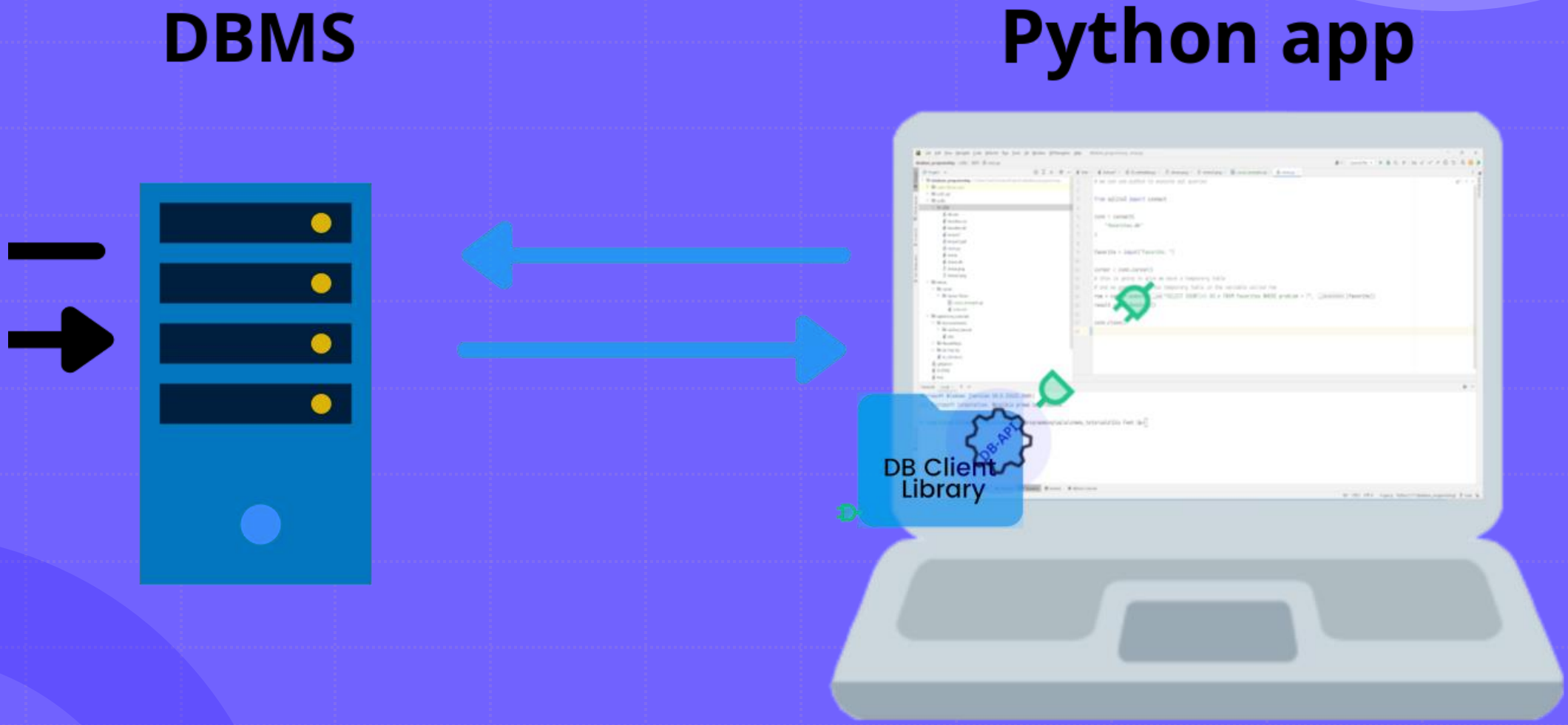
Comment

https://peps.python.org/pep-0248/

Sterowniki

Biblioteki złużące do łączenia się z bazami danych (i tym samym przeważnie implementujące standard DB-API) nazywa się często **sterownikami** (*ang. driver*), **konektorami** (*ang. connector*) lub **adapterami** (*ang. adapater*).

1. DB Client Library (sterownik) implementuje standard DB-API



Popularne sterowniki do MySQL

- **MySQL Connector/Python** https://dev.mysql.com/doc/connector-python/en/
  - mysqlclient https://mysqlclient.readthedocs.io/
  - PyMySQL https://pymysql.readthedocs.io/en/latest/
  - aiomysql https://aiomysql.readthedocs.io/en/stable/
  - asyncmy https://github.com/long2ice/asyncmy
  - CyMySQL https://github.com/nakagami/CyMySQL
  - PyODBC https://github.com/mkleehammer/pyodbc

Popularne sterowniki do PostgreSQL

- psycopg2 https://www.psycopg.org/
- pg8000 https://github.com/tlocke/pg8000
- asyncpg https://magicstack.github.io/asyncpg/current/
- psycopg2cffi https://github.com/chtd/psycopg2cffi

Popularne sterowniki do SQLite

- pysqlite https://docs.python.org/3/library/sqlite3.html
  - aiosqlite https://aiosqlite.omnilib.dev/en/stable/
- pysqlcipher (używa https://www.zetetic.net/sqlcipher/)

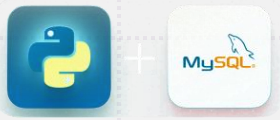
Popularne sterowniki do Microsoft SQL Server

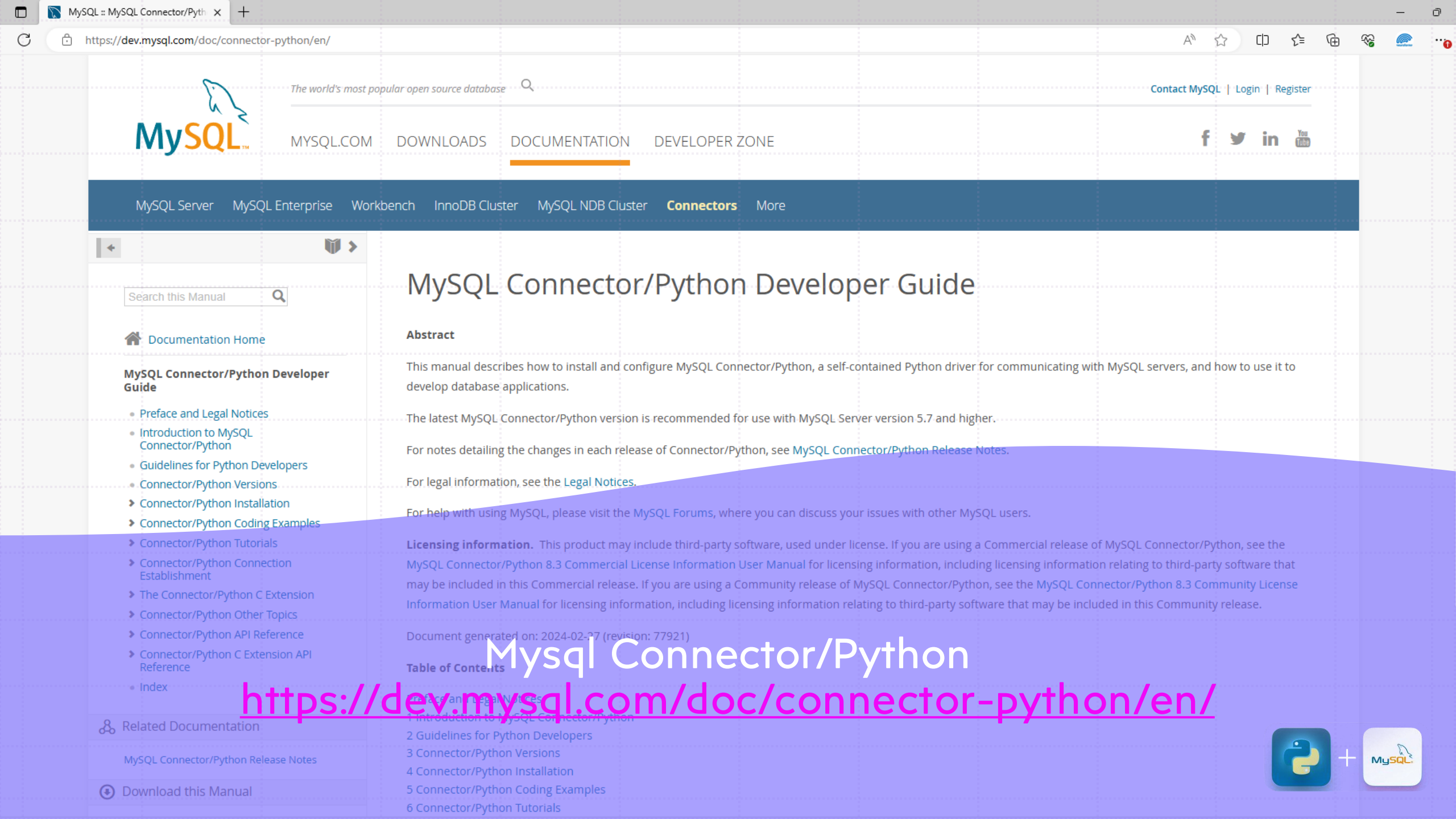
- PyODBC https://github.com/mkleehammer/pyodbc
- mxODBC https://www.egenix.com/
- Pymssql (wrapper wokół https://www.freetds.org/)

Popularne sterowniki do Oracle

- cx_Oracle https://oracle.github.io/python-cx_Oracle/

MySQL w Pythonie




MySQL :: MySQL Connector/Pyth
https://dev.mysql.com/doc/connector-python/en/
The world's most popular open source database
Contact MySQL | Login | Register
MYSQL.COM DOWNLOADS DOCUMENTATION DEVELOPER ZONE
MySQL Server MySQL Enterprise Workbench InnoDB Cluster MySQL NDB Cluster Connectors More
Search this Manual
Documentation Home
MySQL Connector/Python Developer Guide
Preface and Legal Notices
Introduction to MySQL Connector/Python
Guidelines for Python Developers
Connector/Python Versions
Connector/Python Installation
Connector/Python Coding Examples
Connector/Python Tutorials
Connector/Python Connection Establishment
The Connector/Python C Extension
Connector/Python Other Topics
Connector/Python API Reference
Connector/Python C Extension API Reference
Index
Related Documentation
MySQL Connector/Python Release Notes
Download this Manual
MySQL Connector/Python Developer Guide
Abstract
This manual describes how to install and configure MySQL Connector/Python, a self-contained Python driver for communicating with MySQL servers, and how to use it to develop database applications.
The latest MySQL Connector/Python version is recommended for use with MySQL Server version 5.7 and higher.
For notes detailing the changes in each release of Connector/Python, see MySQL Connector/Python Release Notes.
For legal information, see the Legal Notices.
For help with using MySQL, please visit the MySQL Forums, where you can discuss your issues with other MySQL users.
Licensing information. This product may include third-party software, used under license. If you are using a Commercial release of MySQL Connector/Python, see the MySQL Connector/Python 8.3 Commercial License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a Community release of MySQL Connector/Python, see the MySQL Connector/Python 8.3 Community License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Community release.
Document generated on: 2024-02-27 (revision: 77921)
Table of Contents
Preface and Legal Notices
1 Introduction to MySQL Connector/Python
2 Guidelines for Python Developers
3 Connector/Python Versions
4 Connector/Python Installation
5 Connector/Python Coding Examples
6 Connector/Python Tutorials
Mysql Connector/Python
https://dev.mysql.com/doc/connector-python/en/

Sterownik Mysql Connector/Python

Szczegóły DB-API omówimy na przykładzie sterownika **MySQL Connector/Python**

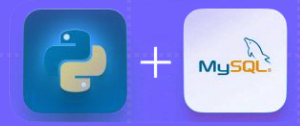
https://dev.mysql.com/doc/connector-python/en/

Większość wprowadzanych zagadnienia będzie jednak wspólna dla wszystkich popularnych (implementujących D-API) sterowników napisanych w Pythonie.

Aktualna wersja sterownika to 8.3 dostępna dla Pythona 3.8+ oraz serwera MySQL 5.6+.

Wersja pobierana z PyPI (MySQL Connector/Python 8.3.0 Community) jest udostępniona na licencji GPLv2

https://downloads.mysql.com/docs/licenses/connector-python-8.3-gpl-en.pdf



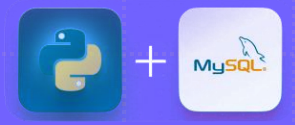
Instalacja sterownika Mysql Connector/Python

Sterownik znajduje się w PyPI pod nazwą mysql-connector-python

https://pypi.org/project/mysql-connector-python/

Aby zainstalować bibliotekę należy w terminalu wykonać komendę:

```
pip install mysql-connector-python
```



Weryfikacja poprawności instalacja sterownika Mysql Connector/Python

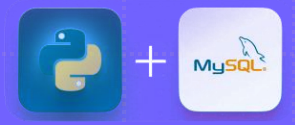
Po zainstalowaniu sterownika należy uruchomić interpreter Pythona

```
python
```

i zaimportować bibliotekę

```
import mysql
```

Jeżeli wywołanie instrukcji nie skończy się wyjątkiem `ModuleNotFoundError` oznacza to, że instalacja przebiegła pomyślnie.



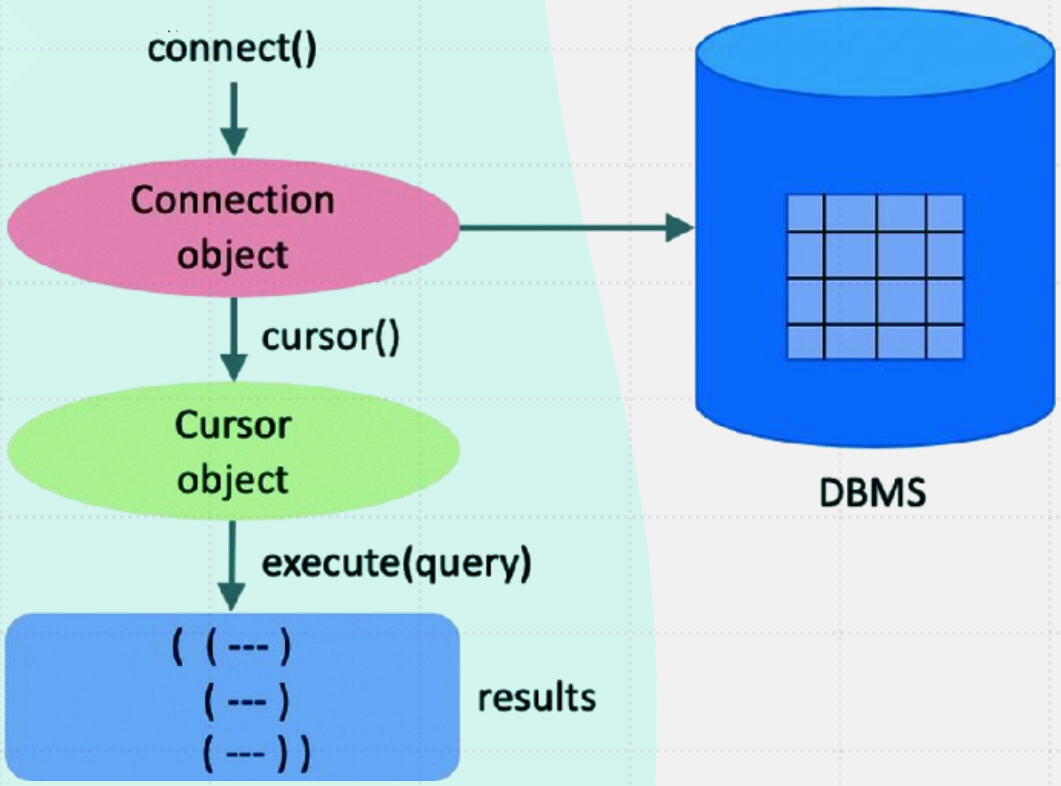
```
mysql.connector
  errorcode
  errors
  connection
  constants
  conversion
  cursor
  dbapi
  locales
    eng
      client_error
  protocol
  utils
```

mysql-connector-python

Struktura pakietu (lista modułów)

DB-API

Technikalia



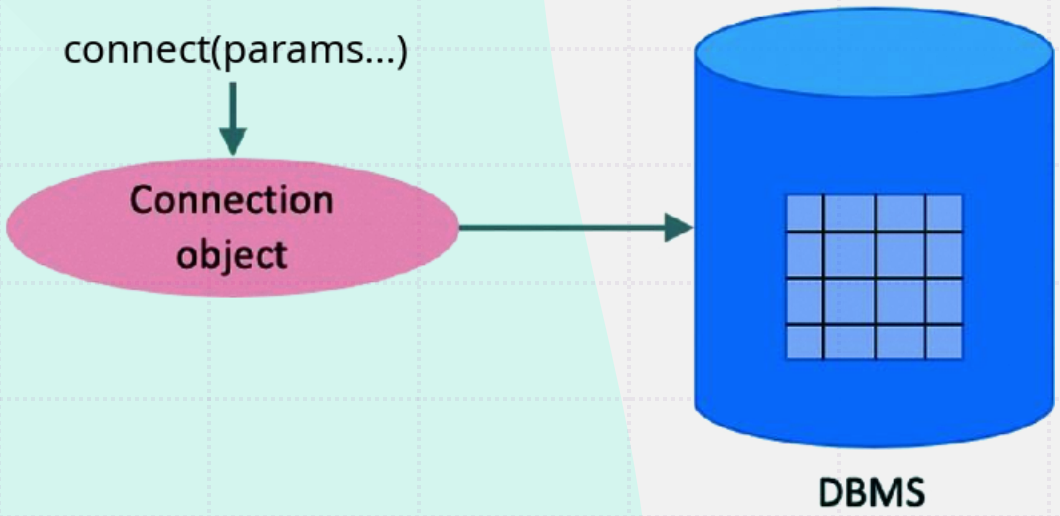
DB-API

MySQL Connector/Python implementuje DB-API w wersji 2.0.

Do podstawowych elementów DB API v2.0 należą:

1. funkcja connect
2. obiekt klasy Connection
3. obiekt klasy Cursor
4. metoda execute obiektu klasy Cursor
5. results

1. Funkcja connect()



Dostęp do bazy realizowany jest za pomocą obiektu klasy Connection. Do tworzenia obietku klasy Connection przeznaczona jest globalnie dostępna funkcja connect(params...). Twórcy DB-API nie zachęcają do stosowanie tradycyjnego konstruktora.

Funkcja **connect(params...)** przyjmuje zestaw parametrów niezbędnych do nawiązania połączenia z bazą, a zwraca obiektu klasy Connection reprezentujący połączenie z bazą.

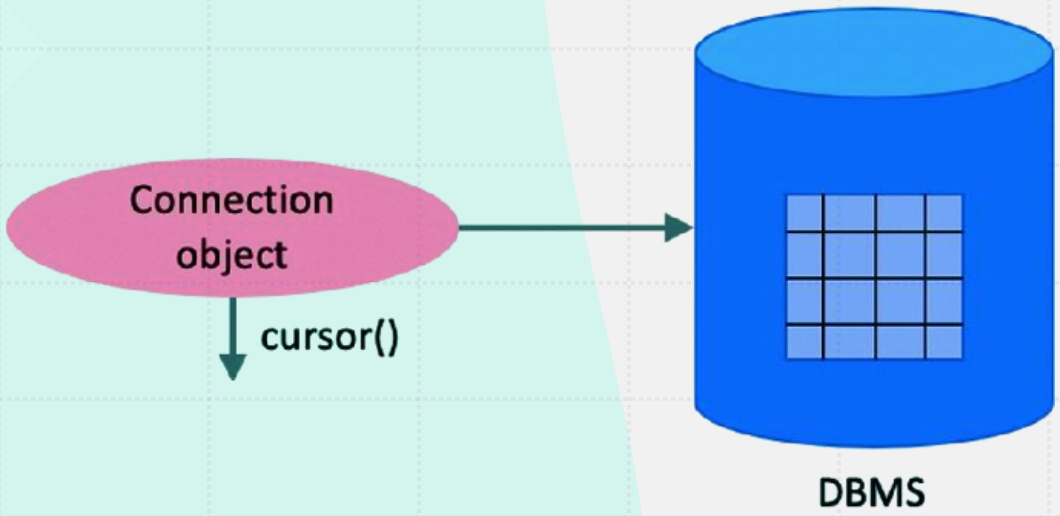
Przykładowy zestaw parametrów połączeniowych:

Parameter	Meaning
`dsn`	Data source name as string
`user`	User name as string (optional)
`password`	Password as string (optional)
`host`	Hostname (optional)
`database`	Database name (optional)

E.g. a connect could look like this:

```
connect(dsn='myhost:MYDB', user='guido', password='234$')
```

2. Obiekt klasy Connection



Interfejs klasy Connection obejmuje metody:

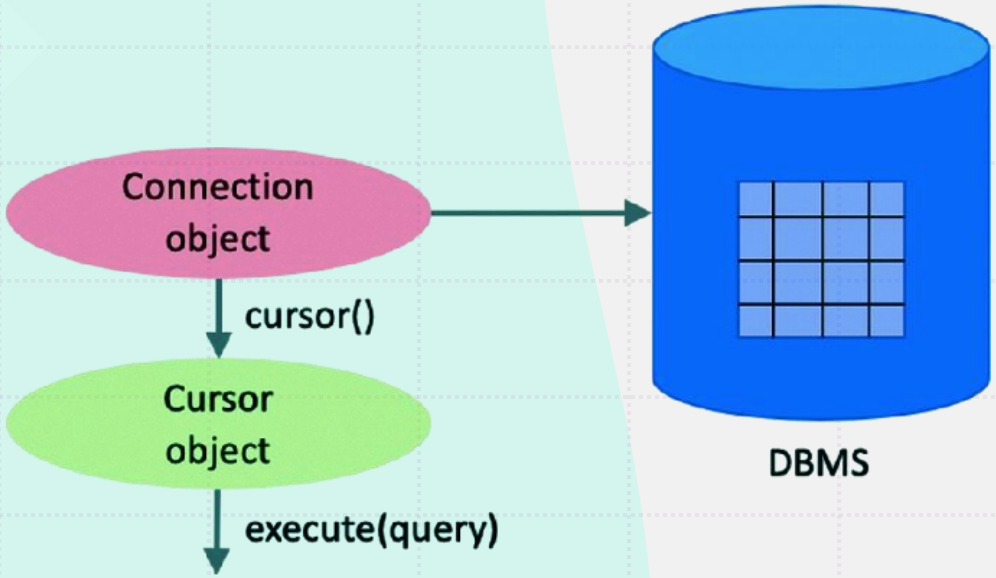
A. **close()** - metoda służąca do zamknięcia połączenia.Zamknięcie połączenia spowoduje cofnięcie wszystkich niezacomitowanych zmian.

B. **commit()** - commituje zmianę (auto-commit powinien być domyślnie wyłączony)

C. **rollback()** - anulowanie transakcji. metoda opcjonalna. Dla baz obsługujących transakcyjność wycofuje zmiany do początku trwającej transakcji. Zamknięcie obiektu bez zacomitowania zmian wywołuje automatycznie tę funkcję.

D. **cursor()** - zwraca obiekt klasy Cursor reprezentujący bazodanowy kursor dla danego połączenia. Jeżeli baza nie wspiera kursorów, moduł powinien zaimplementować zachowania charakterystyczne dla kursora na poziomie pythona.

3. Obiekt klasy Cursor

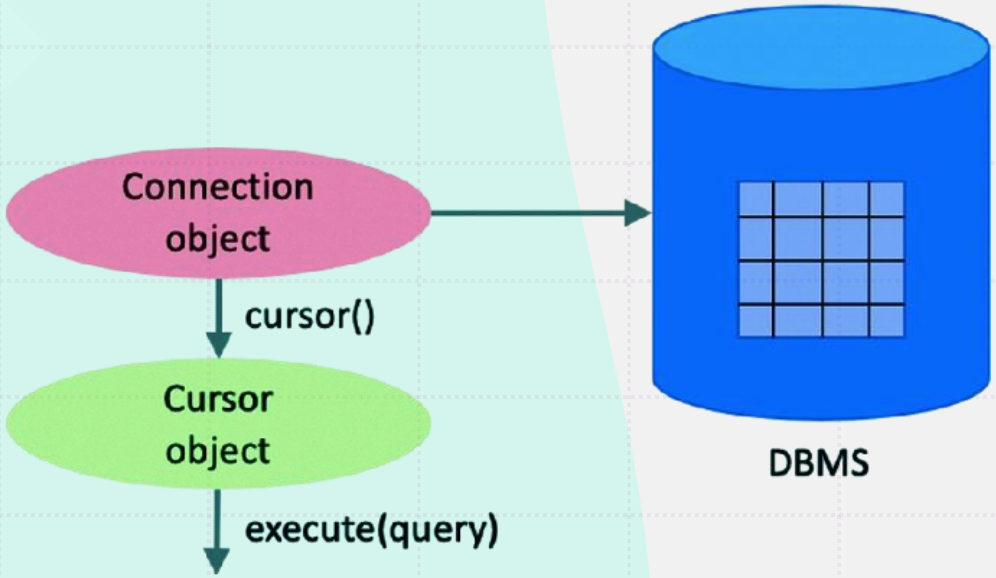


Obiekt klasy Cursor reprezentuje kursor bazodanowy używany do zarządania kontekstem operacji fetch. Cursory tworzone w ramach tego samego połączenia nie są izolowane (tzn. jakiekolwiek zmiany wykonane przez jeden kursor danego połączenia natychmiast są widoczne przez pozostałe kursory tego połączenia. Kursory tworzone w ramach różnych połączeń mogą być (ale nie muszą) izolowane.

Interfejs klasy Cursor obejmuje:

metody	atrybuty
callproc(procname [, parameters])	description
close()	rowcount
execute(operation [, parameters])	arraysize
executemany(operation, seq_of_parameters)	
fetchone()	
fetchmany([size=cursor.arraysize])	
fetchall()	
nextset()	
setinputsizes(sizes)	
setoutputsize(size [, column])	

4. Metoda execute(operation [,parameters]) obiektu klasy Cursor



Najważniejszymi metodami obiektu klasy Cursor są metoda **execute()** oraz rodzina metod fetch(): **fetchone()**, **fetchmany([size=cursor.arraysize])**, **fetchall()**. Metoda execute odpowiada za zbudowanie właściwego zapytania sql oraz wykonanie tego zapytania na bazie. Metody z rodziny fetch odpowiadaja za pobranie części lub całości wyniku. Przyjrzyjmy się bliżej obu metodom.

A. **execute(operation [, parameters])** - metoda tworząca i wykonująca zapytanie na bazie (zapytanie lub instrukcję). Co oznacza tworzenie zapytania?

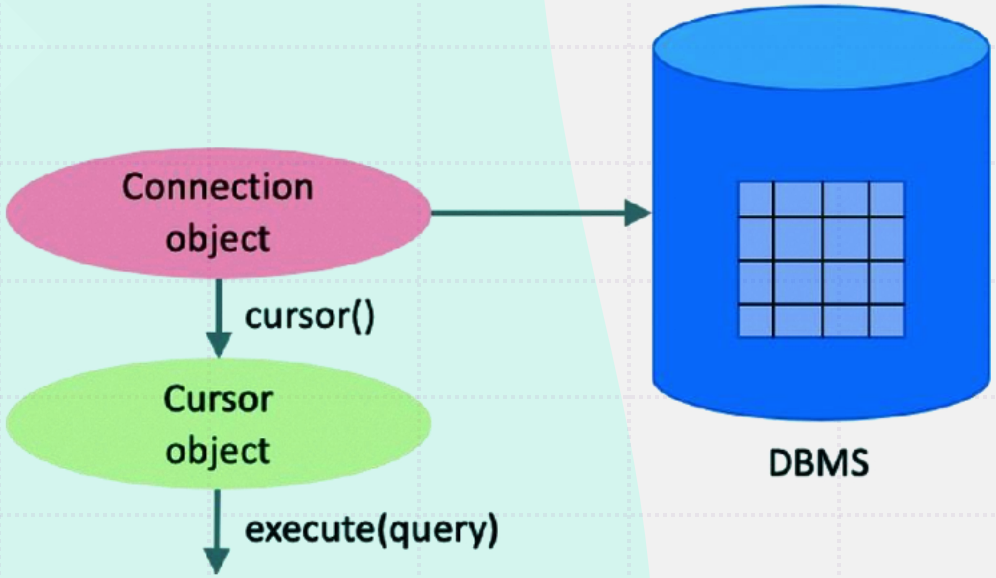
Zapytanie reprezentowane jest napisem. Napis może być sparametryzowany:

```
user_id = 5

f"SELECT * FROM user where id={user_id}"
```

Niestety, taki sposób sparametryzowania zapytania nie chroni nas przed podatnością sql injection (wstrzykiwanie sql). Aby bezpiecznie sforma tować zapytanie należy postępować zgodnie z sygnaturą funkcji. Parametry umieszczamy po przecinku, w wywołaniu metody execute. Parametry mogą być podane w postaci sekwencji lub słownika.

3. Metoda execute(operation [,parameters]) obiektu klasy Cursor

"Always remember to sanitize your input"



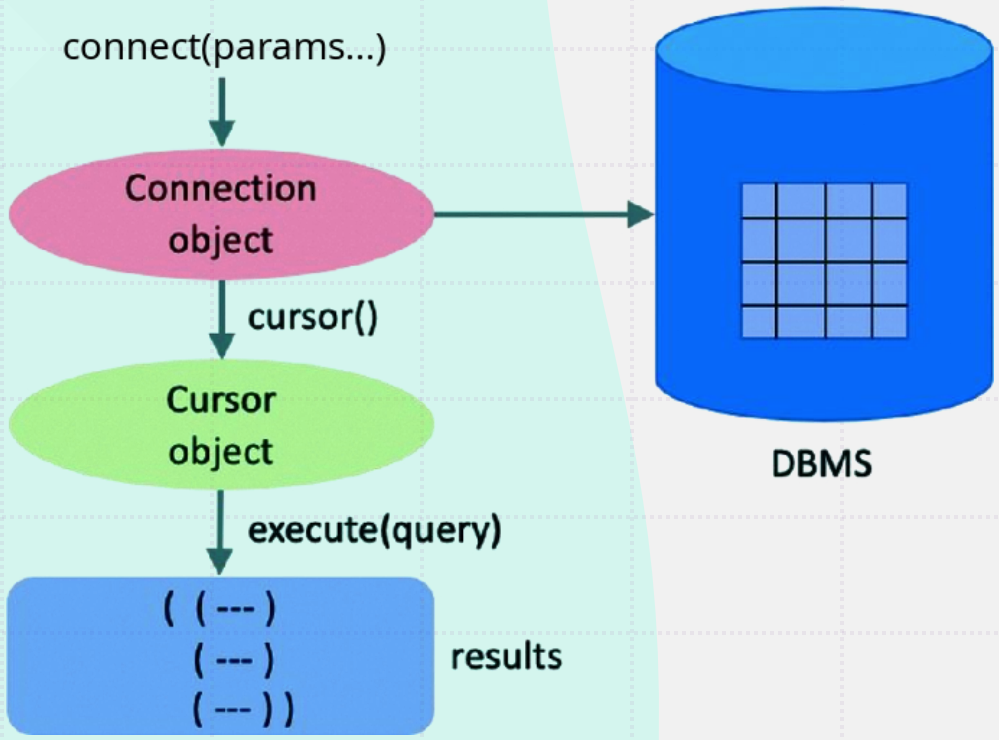
Aby bezpiecznie sformatować zapytanie należy postępować zgodnie z sygnaturą funkcji. Parametry przekazujemy w wywołaniu metody **execute(operation, [,parameters**]), a nie w f-stringu. Parametry mogą być podane w postaci sekwencji lub słownika.

Przekazane w ten sposób parametry zostaną przypisane do "slotów" umieszczonych w napisie reprezentującym operację. Zamiast używać f-stringów napis formatujemy zgodnie z jedną z poniższych konwencji:

paramstyle	Meaning
`qmark`	Question mark style, e.g. `...WHERE name=?`
`numeric`	Numeric, positional style, e.g. `...WHERE name=:1`
`named`	Named style, e.g. `...WHERE name=:name`
`format`	ANSI C printf format codes, e.g. `...WHERE name=%s`
`pyformat`	Python extended format codes, e.g. `...WHERE name=%(name)s`

Za styl formatowania odpowiada globalny parametr **paramstyle**.

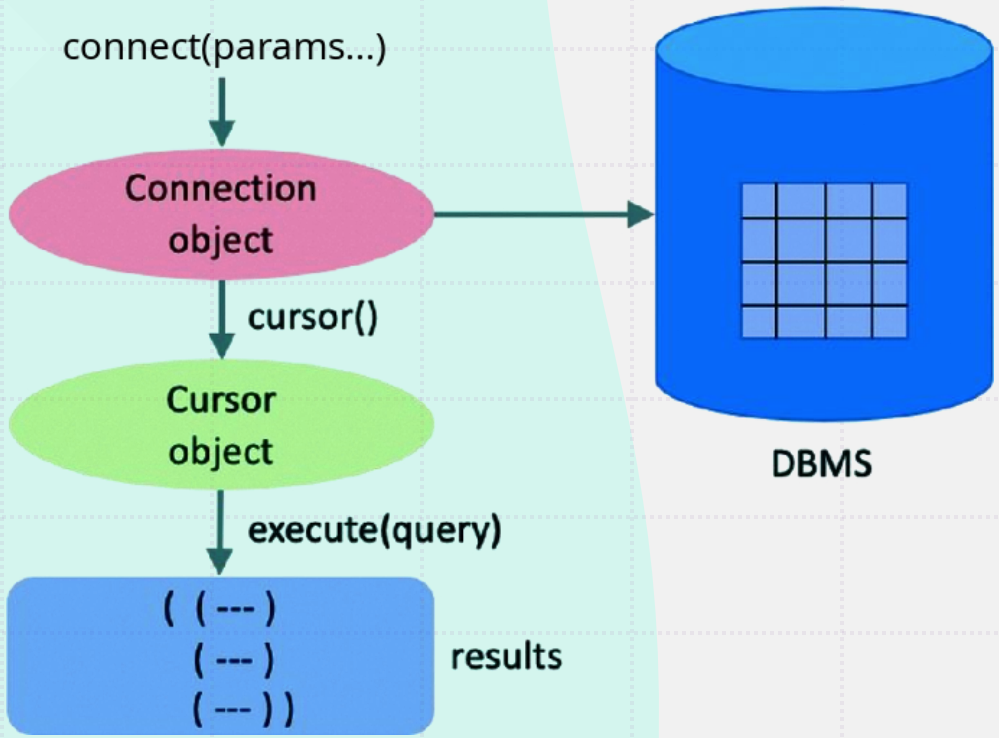
4b. Rodzina metod fetch obiektu klasy Cursor



Motywacja

Result set zwracany w wyniku wykonania zapytania sql potrafi być bardzo duży, często przekraczający rozmiar pamięci RAM. Dlatego podczas pobierania wyniku należy użyć odpowiedniej metody, która pozwoli na pobranie tego result set-a w częściach, fragment po fragmencie, w pętli. Takimi metodami są metody z rodziny fetch.

4b. Rodzina metod fetch obiektu klasy Cursor



A. fetchone() - metoda pobiera z resul-t seta pojdynczy rekord w postaci sekwencji. Po wysyceniu result set-a zwraca None. Jeżeli operacja wykonana przez metodę execute() nie zwróciła żadnego result set-a (jest tak, np. dla operacji: create, drop, insert, update, delete), metoda powinna rzucić wyjątek klasy Error (lub jego podklasę).

B. fetchmany([size=cursor.arraysize]) - metoda pobiera jednorazowo wskazaną w parametrze *size* liczbie rekordów z result set-a. Jeżeli wartość parametry *size* nie jest podana przyjmowana jest domyślna wartość równa wartości atrybutu arraysize obieku klasy Cursor. Zwraca sekwencje sekwencji. Po wysyceniu result set-a zwraca pustą sekwencję.

Jeżeli w result set jest mniej rekordów niż wskazuje parametr size, metoda powinna zwrócić wszystkie dostępne rekordy. Jeżeli operacja wykonana przez metodę execute() nie zwróciła żadnego result set-a (jest tak, np. dla operacji: create, drop, insert, update, delete), metoda powinna rzucić wyjątek klasy Error (lub jego podklasę).

C. fetchall() - metoda zwraca wszystkie (pozostałe) rekordy z result set-a. Zwraca sekwencje sekwencji. Atrybut arraysize obiektu klasy Cursor może mieć wpływ na wynik wywołania tej metody. Mechanizm wyjątków identyczny jak w dwóch poprzednich metodach.

Hierarchia wyjątków w DB-API

DB-API definiuje rodzinę wyjątków, którą powinien posiadać moduły implementujący DB-API.

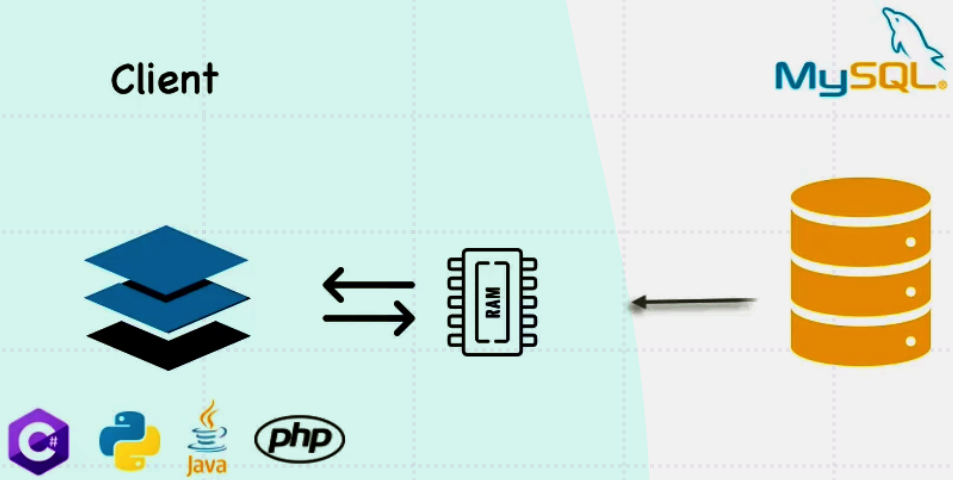
Każdy z wymienionych wyjątków jest opisany w specyfikacji.

```
Exception
|__Warning
|__Error
    |__InterfaceError
    |__DatabaseError
        |__DataError
        |__OperationalError
        |__IntegrityError
        |__InternalError
        |__ProgrammingError
        |__NotSupportedError
```

Rozszerzenia DB-API

Poza omówionymi kwestiami specyfikacja dopuszcze szereg opcjonalnych rozszerzeń dotyczących m.in. metod obiektu klasy Cursor, wyjątków oraz obsługi transakcyjności. Szczegółowy opis tych opcjonalnych rozwiązań można znaleźć w dokumentacji.

Kursor w DB-API jeszcze raz



Klasyczny schemat pobierania danych z bazy zakłada, że pobieramy cały result set za jednym razem i przechowujemy go po stronie klienta w pamięci RAM. W większych bazach danych zawartość pojedynczej tabelki potrafi znacznie przekraczać dostępne obecnie pojemności pamięci RAM. Jeżeli klient nie ma zaimplementowanego jakiegoś mechanizmu buforowania to wykonanie zapytania:

```
SELECT * FROM <duza_tabela>
```

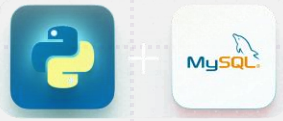
może spowodować przepełnienie dostępnej pamięci RAM i zawieszenie się systemu. Dodatkowo pobranie tak dużej liczby danych z bazy może znacznie spowolnić działanie samej bazy, która musi wygenerować ogromny result set.

Aby uniknąć takiego scenariusza DB-API wymusza na bibliotekach zgodnych z DB-API zaimplementowanie wykonywania zapytań za pomocą kursora. Kursor w sterowanikach implementujących DB-API pełni rolę właśnie takiego mechanizmu buforowania. Opiera się na kursorze bazodanowym. Jeżeli jakaś baza danych nie wspiera kursorów wtedy twórcy bibloteki zgodnej z DB-API są obowiązani samodzielnie zaimplementować w pythonie mechanizm imitujący zachowanie kursora bazodanowego.

MySQL Connector/Python vs DB-API

Sprawdźmy teraz na ile MySQL Connector/Python jest zgodny z DB-API

1. Funkcja connect



Połączenie z serwerem – funkcja connect

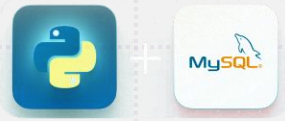
```python
from mysql.connector import connect


cnx = connect(
    user='root',
    password='admin',
)

cnx.close()
```

Funkcja ***connect*** przyjmuje dwa parametry obowiązkowe:

user - użytkownik bazodanowy

password - hasło użytkownika

Wywołanie funkcji *connect* wyłącznie z tymi dwoma parametrami spowoduje nawiązanie połączenie z serwerem MySQL nasłuchującym na adresie 127.0.0.1 (domyślna wartość parametru *host*) na porcie 3306 (domyślna wartość parametru *port*).



Połączenie z bazą - funkcja connect

```python
from mysql.connector import connect


cnx = connect(
    user='root',
    password='admin',
    database='public'
)

cnx.close()
```

Do parametrów opcjonalnych metody connect należą m.in.: *database, host, port, autocommit, …* Pełną listę parametrów metody *connect* można znaleźć na https://dev.mysql.com/doc/connector-python/en/connector-python-connectargs.html

Jeżeli w wywołaniu funkcji connect zostanie podana wartość parametru *database*, połączenie zostanie nawiązane z podaną bazą danych.

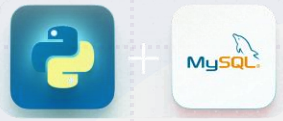
Funkcja connect zwraca obiekt klasy **MySQLConnection**

Połączenie z bazą - klasa MySQLConnection

```python
from mysql.connector.connection import MySQLConnection


cnx = MySQLConnection(
    user='root',
    password='admin',
    database='public'
)

cnx.close()
```

Do nawiązania połączenia z bazą można użyć bezpośrednio klasy *MySQLConnection*, ale w myśl DB-API, preferowanym sposobem nawiązywania połączenia z bazą jest funkcja *connect*.

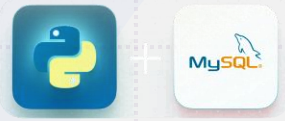
2. Klasa MySQLConnection



```
In [1]: from mysql.connector.connection import MySQLConnection

In [2]: print(list(dir(MySQLConnection)))
['__abstractmethods__', '__annotations__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__', '
__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__red
uce_ex__', '__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__subclasshook__', '__weakref__', '_abc_impl', '_add_default_conn_attrs', '_check_server
_version', '_do_auth', '_do_handshake', '_execute_query', '_get_connection', '_handle_binary_ok', '_handle_binary_result', '_handle_eof', '_handle_load_data_infile',
 '_handle_ok', '_handle_result', '_handle_server_status', '_open_connection', '_post_connection', '_send_cmd', '_send_data', '_validate_callable', '_validate_tls_cip
hersuites', '_validate_tls_versions', 'autocommit', 'can_consume_results', 'charset', 'close', 'cmd_change_user', 'cmd_debug', 'cmd_init_db', 'cmd_ping', 'cmd_proces
s_info', 'cmd_process_kill', 'cmd_query', 'cmd_query_iter', 'cmd_quit', 'cmd_refresh', 'cmd_reset_connection', 'cmd_shutdown', 'cmd_statistics', 'cmd_stmt_close', 'c
md_stmt_execute', 'cmd_stmt_fetch', 'cmd_stmt_prepare', 'cmd_stmt_reset', 'cmd_stmt_send_long_data', 'collation', 'commit', 'config', 'connect', 'connection_id', 'co
nsume_results', 'cursor', 'database', 'disconnect', 'get_row', 'get_rows', 'get_self', 'get_server_info', 'get_server_version', 'get_warnings', 'handle_unread_result
', 'have_next_result', 'in_transaction', 'info_query', 'is_connected', 'is_secure', 'isset_client_flag', 'ping', 'pool_config_version', 'python_charset', 'query_attr
s', 'query_attrs_append', 'query_attrs_clear', 'query_attrs_remove', 'raise_on_warnings', 'reconnect', 'reset_session', 'rollback', 'server_host', 'server_port', 'se
t_allow_local_infile_in_path', 'set_charset_collation', 'set_client_flags', 'set_converter_class', 'set_login', 'set_unicode', 'shutdown', 'sql_mode', 'start_transac
tion', 'time_zone', 'unix_socket', 'unread_result', 'user']
```

Klasa MySQLConnection

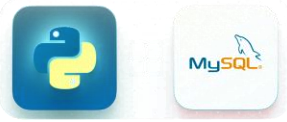
Klasa *MySQLConnection* jest znacznie bogatsza niż to przewiduje interfejs klasy *Connection* DB-API. Metody wymagane przez DB-API podkreślone są na niebiesko.



Klasa MySQLConnection

Klasa *MySQLConnection* dziedziczy po *MySQLConnectionAbstract*.

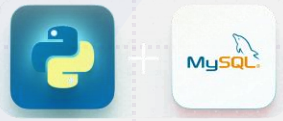
```
ass MySQLConnection(MySQLConnectionAbstract):
    """Connection to a MySQL Server"""

    def __init__(self, **kwargs: Any) -> None:
        self._protocol: Optional[MySQLProtocol] = None
        self._socket: Optional[MySQLSocket] = None
        self._handshake: Optional[HandShakeType] = None
        super().__init__()

        self._converter_class: Type[MySQLConverter] = MySQLConverter

        self._client_flags: int = ClientFlag.get_default()
        self._charset_id: int = 45
        self._sql_mode: Optional[str] = None
        self._time_zone: Optional[str] = None
        self._autocommit: bool = False

        self._user: str = ""
        self._password: str = ""
        self._database: str = ""
        self._host: str = "127.0.0.1"
        elf._port: int = 3306
        lf._unix_socket: Optional[str] = None
        f._client_host: str = ""
        f._client_port: int = 0
        ._ssl: Dict[str, Optional[Union[str, bool, List[str]]]] = {}
        ._force_ipv6: bool = False

        ._use_unicode: bool = True
        get_warnings: bool = False
```



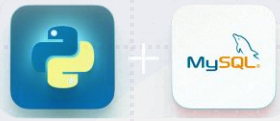
```
In [1]: from mysql.connector.connection import MySQLConnection

In [2]: print(list(dir(MySQLConnection)))
['__abstractmethods__', '__annotations__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__', '
__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__red
uce_ex__', '__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__subclasshook__', '__weakref__', '_abc_impl', '_add_default_conn_attrs', '_check_server
_version', '_do_auth', '_do_handshake', '_execute_query', '_get_connection', '_handle_binary_ok', '_handle_binary_result', '_handle_eof', '_handle_load_data_infile',
 '_handle_ok', '_handle_result', '_handle_server_status', '_open_connection', '_post_connection', '_send_cmd', '_send_data', '_validate_callable', '_validate_tls_cip
hersuites', '_validate_tls_versions', 'autocommit', 'can_consume_results', 'charset', 'close', 'cmd_change_user', 'cmd_debug', 'cmd_init_db', 'cmd_ping', 'cmd_proces
s_info', 'cmd_process_kill', 'cmd_query', 'cmd_query_iter', 'cmd_quit', 'cmd_refresh', 'cmd_reset_connection', 'cmd_shutdown', 'cmd_statistics', 'cmd_stmt_close', 'c
md_stmt_execute', 'cmd_stmt_fetch', 'cmd_stmt_prepare', 'cmd_stmt_reset', 'cmd_stmt_send_long_data', 'collation', 'commit', 'config', 'connect', 'connection_id', 'co
nsume_results', 'cursor', 'database', 'disconnect', 'get_row', 'get_rows', 'get_self', 'get_server_info', 'get_server_version', 'get_warnings', 'handle_unread_result
', 'have_next_result', 'in_transaction', 'info_query', 'is_connected', 'is_secure', 'isset_client_flag', 'ping', 'pool_config_version', 'python_charset', 'query_attr
s', 'query_attrs_append', 'query_attrs_clear', 'query_attrs_remove', 'raise_on_warnings', 'reconnect', 'reset_session', 'rollback', 'server_host', 'server_port', 'se
t_allow_local_infile_in_path', 'set_charset_collation', 'set_client_flags', 'set_converter_class', 'set_login', 'set_unicode', 'shutdown', 'sql_mode', 'start_transac
tion', 'time_zone', 'unix_socket', 'unread_result', 'user']
```

Klasa MySQLConnection

Na czerwono podkreślone są metody i atrybuty klasy *MySQLConnection* wymagane przez interfejs klasy *MySQLConnectionAbstract*.



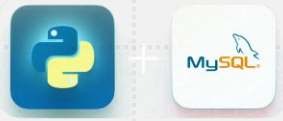
2b. Obsługa wyjątków



```python
import mysql.connector
from mysql.connector import errorcode


try:
    cnx = mysql.connector.connect(
        user='root',
        password='admin',
        database='public'
    )
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        print("Something is wrong with your username or password")
    elif err.errno == errorcode.ER_BAD_DB_ERROR:
        print("Database does not exists")
    else:
        print(err)
else:
    cnx.close()
```

Połączenie z bazą - obsługa wyjątków

Istnieje kilka powodów, dla których połączenie z bazą danych może skończyć się niepowodzeniem. Do najczęstszych należą: nieprawidłowe poświadczenia, nieprawidłowa nazwa bazy danych, nieprawidłowy adres serwera. W takich przypadkach próba nawiązania połączenia z bazą zakończy się błędem. W naszym kodzie powinniśmy takie błędy obsłużyć.



```
In [1]: from mysql.connector import errors

In [2]: print(dir(errors))
['DataError', 'DatabaseError', 'Dict', 'Error', 'ErrorClassTypes', 'ErrorTypes', 'IntegrityError', 'InterfaceError', 'InternalError', 'Mapping', '
NotSupportedError', 'OperationalError', 'Optional', 'PoolError', 'ProgrammingError', 'StrOrBytes', 'Tuple', 'Type', 'Union', 'Warning', '_CUSTOM_E
RROR_EXCEPTIONS', '_ERROR_EXCEPTIONS', '_SQLSTATE_CLASS_EXCEPTION', '__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loa
der__', '__name__', '__package__', '__spec__', 'custom_error_exception', 'get_client_error', 'get_exception', 'get_mysql_exception', 'read_bytes',
 'read_int']
```

Błędy

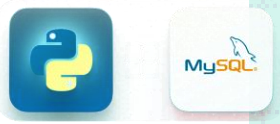
Klasy wyjątków obsługiwane przez driver można znaleźć w module *mysql.connector.errors*.
Wyjątki wymagane przez DB-API podkreślone są na niebiesko.

Klasy wyjątków DB-API

Zestawienie (z modułu *mysq.connector.errors*) przedstawia mapowanie klas wyjątków DB-API zaimplementowanych w *mysql-connector-python* na klasy SQLSTATE *MySQL Server*.

Widzimy na przykład, że klasa *ProgrammingError* reprezentuje SQLSTATE klas: 24, 25, 26, 27, 28, 2A, 2C, 34 ,35, 37, 3C, 3D, 3F, 42

```
_SQLSTATE_CLASS_EXCEPTION: Dict[str, ErrorClassTypes] = {
    "02": DataError,  # no data
    "07": DatabaseError,  # dynamic SQL error
    "08": OperationalError,  # connection exception
    "0A": NotSupportedError,  # feature not supported
    "21": DataError,  # cardinality violation
    "22": DataError,  # data exception
    "23": IntegrityError,  # integrity constraint violation
    "24": ProgrammingError,  # invalid cursor state
    "25": ProgrammingError,  # invalid transaction state
    "26": ProgrammingError,  # invalid SQL statement name
    "27": ProgrammingError,  # triggered data change violation
    "28": ProgrammingError,  # invalid authorization specification
    "2A": ProgrammingError,  # direct SQL syntax error or access rule violation
    "2B": DatabaseError,  # dependent privilege descriptors still exist
    "2C": ProgrammingError,  # invalid character set name
    "2D": DatabaseError,  # invalid transaction termination
    "2E": DatabaseError,  # invalid connection name
    "33": DatabaseError,  # invalid SQL descriptor name
    "34": ProgrammingError,  # invalid cursor name
    "35": ProgrammingError,  # invalid condition number
    "37": ProgrammingError,  # dynamic SQL syntax error or access rule violation
    "3C": ProgrammingError,  # ambiguous cursor name
    "3D": ProgrammingError,  # invalid catalog name
    "3F": ProgrammingError,  # invalid schema name
    "40": InternalError,  # transaction rollback
    "42": ProgrammingError,  # syntax error or access rule violation
    "44": InternalError,  # with check option violation
    "HZ": OperationalError,  # remote database access
    "XA": IntegrityError,
    "0K": OperationalError,
    "HY": DatabaseError,  # default when no SQLState provided by MySQL server
}
```

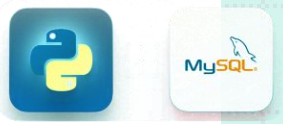


Kod błędu

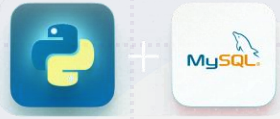
Każdy wyjątek mysql-connector-python posiada atrybut errno reprezentujący kod błędu MySQL.

Mapowanie kodów błędu na symbole błędów znajduje się w module *mysql.connector.errorcode*

```
OBSOLETE_ER_DISK_FULL = 1021
ER_DUP_KEY = 1022
OBSOLETE_ER_ERROR_ON_CLOSE = 1023
ER_ERROR_ON_READ = 1024
ER_ERROR_ON_RENAME = 1025
ER_ERROR_ON_WRITE = 1026
ER_FILE_USED = 1027
OBSOLETE_ER_FILSORT_ABORT = 1028
OBSOLETE_ER_FORM_NOT_FOUND = 1029
ER_GET_ERRNO = 1030
ER_ILLEGAL_HA = 1031
ER_KEY_NOT_FOUND = 1032
ER_NOT_FORM_FILE = 1033
ER_NOT_KEYFILE = 1034
ER_OLD_KEYFILE = 1035
ER_OPEN_AS_READONLY = 1036
ER_OUTOFMEMORY = 1037
ER_OUT_OF_SORTMEMORY = 1038
OBSOLETE_ER_UNEXPECTED_EOF = 1039
ER_CON_COUNT_ERROR = 1040
ER_OUT_OF_RESOURCES = 1041
ER_BAD_HOST_ERROR = 1042
ER_HANDSHAKE_ERROR = 1043
ER_DBACCESS_DENIED_ERROR = 1044
ER_ACCESS_DENIED_ERROR = 1045
ER_NO_DB_ERROR = 1046
ER_UNKNOWN_COM_ERROR = 1047
ER_BAD_NULL_ERROR = 1048
ER_BAD_DB_ERROR = 1049
ER_TABLE_EXISTS_ERROR = 1050
ER_BAD_TABLE_ERROR = 1051
ER_NON_UNIQ_ERROR = 1052
ER_SERVER_SHUTDOWN = 1053
```



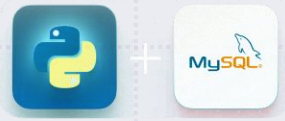
2c. Klasa MySQLConnection jako menadżer kontekstu



MySQLConnection

Klasa *MySQLConnection* implementuje protokół menadżera kontektu w związku z tym obiekt klasy *MySQLConnection* jest menadżerem kontekstu (*ang. Context Manager*).

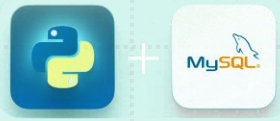
```python
class MySQLConnectionAbstract(ABC):
    """Abstract class for classes connecting to a MySQL server."""
...

    def __enter__(self) -> MySQLConnectionAbstract:
        return self

    def __exit__(
        self,
        exc_type: Type[BaseException],
        exc_value: BaseException,
        traceback: TracebackType,
    ) -> None:
        self.close()

    def get_self(self) -> MySQLConnectionAbstract:
        """Returns self for `weakref.proxy`.

        This method is used when the original object is needed when using
        `weakref.proxy`.
        """
        return self

    @property
```



MySQLConnection

Klasa MySQLConnection ma zaimplementowany protokół menadżera kontekstu, więc zamiast ręcznie zamykać połączenie można użyć słówka kluczowego *with*.

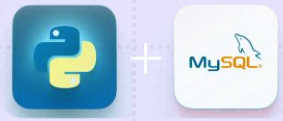
```
from mysql.connector import connect


with connect(user='root', password='admin', database='public') as cnx:
    ...
```



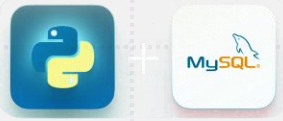
MySQLConnection

Użycie obiektu klasy MySQLConnection jako menadżera kontekstu.

```python
import mysql.connector
from mysql.connector import errorcode


try:
    with mysql.connector.connect(
        user='root',
        password='admin',
        database='public'
    ) as cnx:
        ...
except mysql.connector.Error as err:
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        print("Something is wrong with your username or password")
    elif err.errno == errorcode.ER_BAD_DB_ERROR:
        print("Database does not exists")
    else:
        print(err)
```



3. Cursor

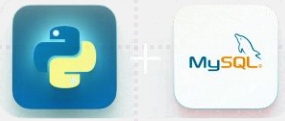


Cursor

```python
import mysql.connector


cnx = mysql.connector.connect(
    user='root',
    password='admin',
    database='public'
)
cursor = cnx.cursor()

cursor.close()
cnx.close()
```

Metoda cursor() obiektów klasy MySQLConnection zwraca obiekt klasy MySQLCursor.

(obiekt klasy MySQLConnection po raz ostatni użyty nie jako menadżer kontekstu)

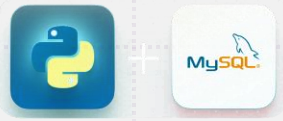


Cursor

```python
import mysql.connector


with mysql.connector.connect(
    user='root',
    password='admin',
    database='public'
) as cnx:
    cursor = cnx.cursor()
    cursor.close()
```

Metoda cursor() obiektów klasy MySQLConnection zwraca obiekt klasy MySQLCursor.

(obiekt klasy MySQLConnection użyty jako menadżer kontekstu)



Cursor

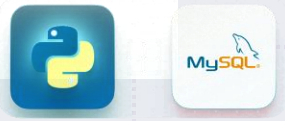
Klasa *MySQLCursorAbstract* również implementuje protokół menadżera kontektu w związku z tym obiekt klasy *MySQLCursor* jest menadżerem kontekstu.

```python
class MySQLCursorAbstract(ABC):
    """Abstract cursor class

    Abstract class defining cursor class with method and members
    required by the Python Database API Specification v2.0.
    """

    def __init__(self) -> None:
        """Initialization"""
        self._description: Optional[List[DescriptionType]] = None
        self._rowcount: int = -1
        self._last_insert_id: Optional[int] = None
        self._warnings: Optional[List[WarningType]] = None
        self._warning_count: int = 0
        self._executed: Optional[bytes] = None
        self._executed_list: List[StrOrBytes] = []
        self._stored_results: List[MySQLCursorAbstract] = []
        self.arraysize: int = 1

    def __enter__(self) -> MySQLCursorAbstract:
        return self

    def __exit__(
        self,
        exc_type: Type[BaseException],
        exc_value: BaseException,
        traceback: TracebackType,
    ) -> None:
        self.close()
```

Cursor

Klasa *MySQLCursor* implementuje protokół iteratora w związku z tym po obiektach klasy *MySQLCursor* można iterować.

```
class MySQLCursor(CursorBase):
...

    def __iter__(self) -> Iterator[RowType]:
        """
        Iteration over the result set which calls self.fetchone()
        and returns the next row.
        """
        return iter(self.fetchone, None)
...


    def __next__(self) -> RowType:
        """
        Used for iterating over the result set. Calles self.fetchone()
        to get the next row.
        """
        try:
            row = self.fetchone()
        except InterfaceError:
            raise StopIteration from None
        if not row:
            raise StopIteration
        return row

    def close(self) -> bool:
        """Close the cursor

        Returns True when successful, otherwise False.
        """
        if self._connection is None:
```



Cursor

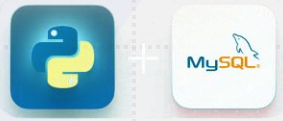
Użycie obiektu klasy *MySQLCursor* jako menadżera kontekstu.

```
import mysql.connector


with mysql.connector.connect(
    user='root',
    password='admin',
    database='public'
) as cnx:
    with cnx.cursor() as cursor:
        ...
```

```
In [3]: print(dir(cursor))
['__abstractmethods__', '__annotations__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__for
mat__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__mo
dule__', '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__subcl
asshook__', '__weakref__', '_abc_impl', '_affected_rows', '_batch_insert', '_buffered', '_check_executed', '_cnx', '_description', '_execute_iter'
, '_executed', '_executed_list', '_fetch_warnings', '_handle_eof', '_handle_result', '_handle_resultset', '_handle_warnings', '_last_insert_id', '
_nextrow', '_raw', '_raw_as_string', '_rowcount', '_stored_results', '_warning_count', '_warnings', 'add_attribute', 'arraysize', 'callproc', 'cle
ar_attributes', 'close', 'column_names', 'description', 'execute', 'executemany', 'fetchall', 'fetchmany', 'fetchone', 'fetchwarnings', 'get_attri
butes', 'lastrowid', 'nextset', 'remove_attribute', 'reset', 'rowcount', 'setinputsizes', 'setoutputsize', 'statement', 'stored_results', 'warning
_count', 'warnings', 'with_rows']
```

Klasa MySQLCursor

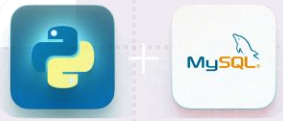
Na niebiesko podkreślone są metody wymagane przez DB-AP.



Cursor

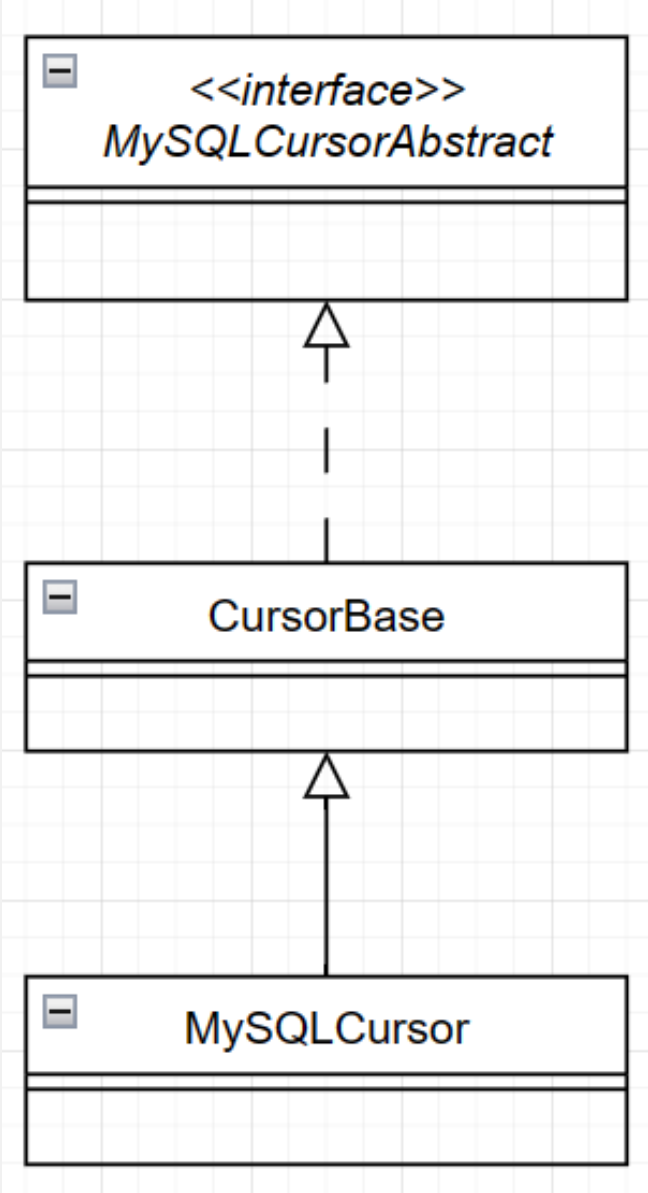
1. Sygnatura metody cursor obiektu MySQLConnection

```
def cursor(
    self,
    buffered: Optional[bool] = None,
    raw: Optional[bool] = None,
    prepared: Optional[bool] = None,
    cursor_class: Optional[Type[MySQLCursor]] = None,  # type: ignore[override]
    dictionary: Optional[bool] = None,
    named_tuple: Optional[bool] = None,
) -> MySQLCursor:
    """Instantiates and returns a cursor
```

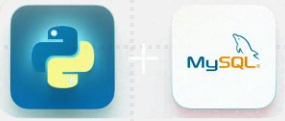


Cursor

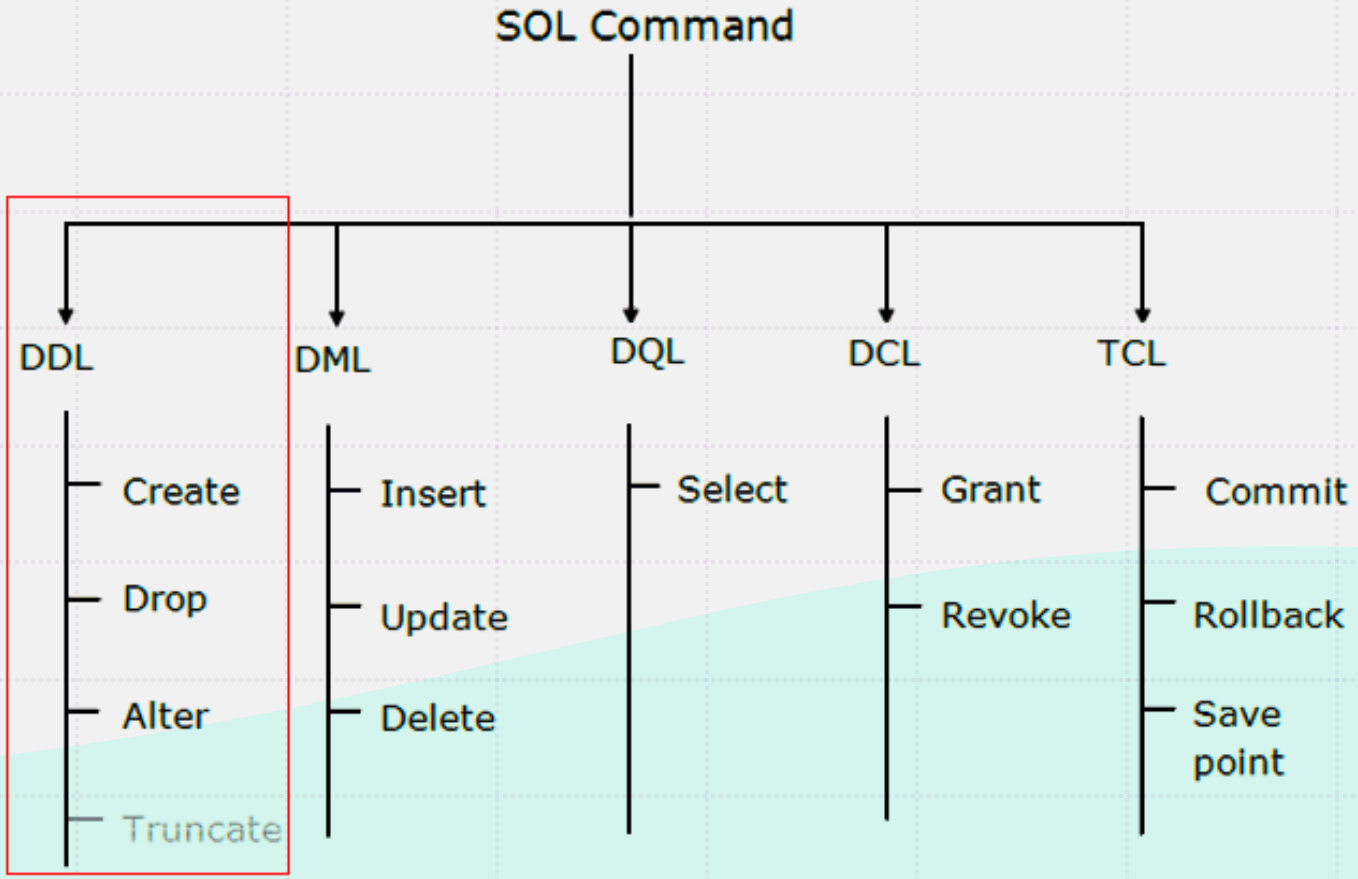
1. Klasa MySQLCursor rozszerza klasę *CursorBase*. Klasa *CursorBase* zawiera wszystkie składowe klasy Cursor z DB-API i implementuje abstrakcyjną klasę My*SQLCursorAbstract pełniącą rolę interfejsu*.



4. Metoda execute obiektu klasy Cursor



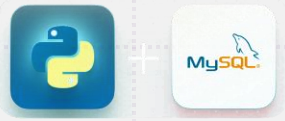
Instrukcje DDL



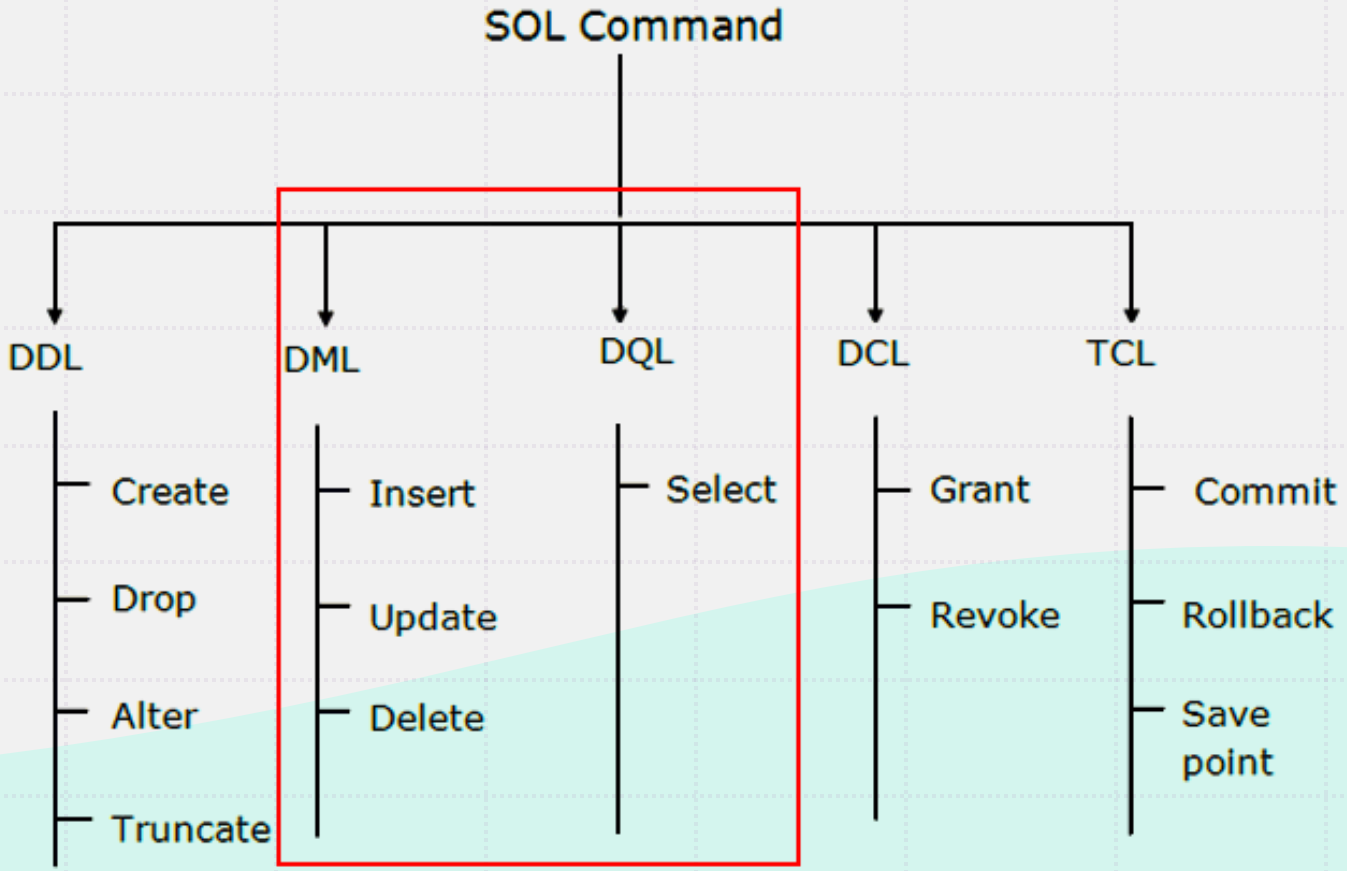
Instrukcje DDL

```
with connect(
    user=USERNAME,
    password=PASSWORD,
    database=DB
) as cnx:
    with cnx.cursor() as cursor:
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS client (
            id INT AUTO_INCREMENT PRIMARY KEY,
            first_name VARCHAR(50) NOT NULL,
            last_name VARCHAR(100) NOT NULL,
            email VARCHAR(100) UNIQUE
            );
        """)
```

Instrukcje DDL nie wymagają wywołania metody commit



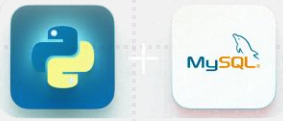
Instrukcje DML I DQL



Instrukcje DML

Instrukcje DML wymagają wywołania metody commit

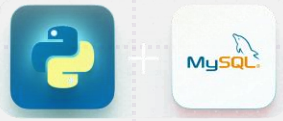
```python
import mysql.connector

with mysql.connector.connect(
    user='root',
    password='admin',
    database='public',
) as cnx:
    with cnx.cursor() as cursor:
        cursor.execute("""
            INSERT INTO client(first_name, last_name, email)
            VALUE
            ('John', 'Doe', 'john.doe@example.com');
        """)
        cnx.commit()
```



Instrukcje DQL

Metoda fetchall() - result set w formacie listy tupli

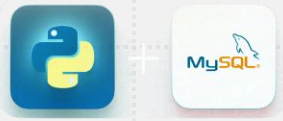
```python
import mysql.connector

with mysql.connector.connect(
    user='root',
    password='admin',
    database='public',
) as cnx:
    with cnx.cursor() as cursor:
        cursor.execute("""SELECT * FROM client;""")

        result = cursor.fetchall()  # list of tuples
```



Instrukcje DQL

Metoda fetchall() - result set w formacie listy słowników

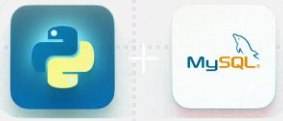
```python
import mysql.connector

with mysql.connector.connect(
    user='root',
    password='admin',
    database='public',
) as cnx:
    with cnx.cursor(dictionary=True) as cursor:
        cursor.execute("""SELECT * FROM client;""")

        result = cursor.fetchall()  # list of dicts
```



Instrukcje DQL

Metoda fetchone()

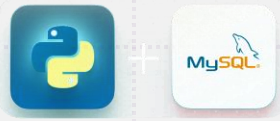
```python
import mysql.connector

with mysql.connector.connect(
    user='root',
    password='admin',
    database='public',
) as cnx:
    with cnx.cursor() as cursor:
        cursor.execute("""SELECT * FROM client;""")

        row = cursor.fetchone()
        while row is not None:
            print(row)
            row = cursor.fetchone()
```



Instrukcje DQL

Metoda fetchmany()

```python
import mysql.connector

with mysql.connector.connect(
    user='root',
    password='admin',
    database='public',
) as cnx:
    with cnx.cursor() as cursor:
        cursor.execute("""SELECT * FROM client;""")

        batch = cursor.fetchmany(size=3)
        while batch:  # batch = [] if end of records
            for row in batch:
                print(row)
            print("End of batch")
            batch = cursor.fetchmany(size=3)
```



Instrukcje DQL

Obiekt klasy MySQLCursor jako iterator

```python
import mysql.connector

with mysql.connector.connect(
    user='root',
    password='admin',
    database='public',
) as cnx:
    with cnx.cursor() as cursor:
        cursor.execute("""SELECT * FROM client;""")

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