BAZY DANYCH

Plan szkolenia

Plan modułu

TEORIA

1 **BAZY DANYCH**

- DB API 2.0
- przegląd popularnych “connectorów” dla RDBMS
- obsługa zapytań
- połączenie z bazami nierelacyjnymi
- ORM w Pythonie

Systemy zarządzania relacyjnymi bazami danych

- **Systemy zarządzania relacyjnymi bazami danych** (*RDBMS – Relational Database Management System*), jak np. PostgreSQL lub Oracle, oferują zaawansowane podejście do przechowywania, wyszukiwania i odzyskiwania trwałych danych
- Relacyjne bazy danych wykorzystują różne dialekty SQL
- Pomimo istnienia standardów SQL, żadne dwa RDBMS nie implementują dokładnie tego samego dialektu SQL

Standard DB API 2.0

- Standardowa biblioteka Python'a nie jest wyposażona w interfejs RDBMS (wyjątkiem jest moduł *sqlite3*, który jest pełną implementcją, a nie tylko interfejsem)
- Jednak wiele modułów firm trzecich pozwala programom Python'a uzyskać dostęp do określonego RDBMS
- Takie moduły są w większości zgodne ze standardem **Python Database API v2.0**, znanym również jako *DB API v2.0* (☞ PEP 249)

Standard DB API 2.0

BAZA DANYCH	MODUŁ PYTHONA	PROJEKT/DOKUMENTACJA
MySQL	*MySQL Connector/Python*	☞ link
	mysqlclient	☞ link
	pymysql	☞ link
PostgreSQL	*psycopg2*	☞ link
	PyGreSQL	☞ link
	py-postgresql	☞ link
	pg8000	☞ link
Oracle	*cx_Oracle*	☞ link

Nawiązanie połączenia

- Po zaimportowaniu dowolnego modułu zgodnego z DB API należy wywołać funkcję *connect* z parametrami specyficznymi dla bazy danych

Nawiązanie połączenia

```
connect(parametry...)
```

- Opcjonalne parametry:

database	nazwa bazy danych do której się podłączamy
dsn	nazwa używanego źródła danych
host	nazwa maszyny na której działa baza danych
user	nazwa użytkownika używanego do połączenia
password	hasło używane do połączenia

Nawiązanie połączenia – przykłady

Połączenie do MySQL

```
# zainstalowany moduł mysql-connector-python
import mysql.connector
conn = mysql.connector.connect(host='localhost',
                               user='root',
                               password='admin',
                               database='magazyn'
                               charset='utf8mb4')
```

Połączenie do PostgreSQL

```
import psycopg2

conn = psycopg2.connect(host="localhost",
                        user="postgres",
                        password="admin",
                        port="5432",
                        database="magazyn")
```

Operacje na obiekcie połączenia

- Funkcja *connect* zwraca instancję *Connection*, która reprezentuje połączenie z bazą danych
- Instancja połączenia dostarcza metod:

METODA	DZIAŁANIE
close()	zamyka połączenie
commit()	zatwierdza oczekujące transakcje
rollback()	wycofuje zmiany dokonane w ramach transakcji do punktu początkowego zamknięcie połączenia bez zatwierdzenia zmian spowoduje niejawnie wycofanie zmian
cursor()	zwraca obiekt kursora (instancję klasy *Cursor*) obiekt reprezentuje kursor bazy danych, który jest używany do zarządzania kontekstem operacji pobierania

Operacje na kursorze

- Kursor dostarcza metody i atrybuty używane do operacji na bazie danych

ATRYBUT	ZNACZENIE
description	sekwencja zawierająca 7-elementowe krotki każda krotka zawiera informacje opisujące jedną kolumnę wynikową (*name*, *type_code*, *display_size*, *internal_size*, *precision*, *scale*, *null_ok*)
rowcount	liczba wierszy, które zostały zwrócone (operacje DQL, jak np. *select*) lub zmodyfikowane/utworzone (operacje DML, jak np. *update*, czy *insert*)

Operacje na kursorze

Metody kursora

```
callproc(procname [, parameters])
close()
```

METODA	DZIAŁANIE
callproc	wywołuje procedurę składowaną wyniki są zwracane poprzez zmodyfikowane kopie parametrów wejściowych
close	powoduje zamknięcie kursora

Operacje na kursorze

Metody kursora

```
execute(operation [, parameters])
executemany(operation, seq_of_parameters)
```

METODA	DZIAŁANIE
execute	przygotowuje i wykonuje operacje na bazie (kwerendę lub polecenie) aby wstawić wiele wierszy można jako parametry przekazać listy krotek lub użyć polecenia *executemany()*
executemany	przygotowuje operację (kwerendę lub polecenia) i ją wykonuje z podanymi sekwencjami parametrów lub mapowaniami podobne działanie do wielokrotnych wywołań *execute()*

Operacje na kursorze

Metody kursora

```
fetchone()
fetchmany([size=curs.arraysize])
fetchall()
```

METODA	DZIAŁANIE
fetchone	pobiera następny wiersz zestawu wyników zapytania, zwracając pojedynczą sekwencję lub *None*, gdy nie ma już dostępnych danych
fetchmany	pobiera następny zestaw wierszy wyniku zapytania, zwracając sekwencję sekwencji (np. listę krotek) gdy nie ma już dostępnych wierszy zwracana jest pusta sekwencja
fetchall	pobiera wszystkie (lub pozostałe) wiersze wyniku zapytania, zwracając je jako sekwencję sekwencji (np. listę krotek)

Styl parametrów

- Moduł zgodny z DB API posiada atrybut *paramstyle*, który określa styl znaczników używanych jako symbole zastępcze parametrów

Atrybut *paramstyle*

```
select = 'SELECT * FROM TABELA WHERE '

c.execute(select + 'KOL=%s', (wart,))                    # format
c.execute(select + 'KOL=:param', {'param': wart})        # named
c.execute(select + 'KOL=:1', (wart,))                    # numeric
c.execute(select + 'KOL=%(param)s', {'param': wart})     # pyformat
c.execute(select + 'KOL=?', (wart,))                     # qmark
```

- W ten sposób można tworzyć szablony zapytań

Przykład

Przykład

```
from sqlite3 import connect

# podłączenie się do istniejącej bazy danych
conn = connect(r'C:\db\osoby.db')

# utworzenie tabeli
cursor = conn.cursor()
cursor.execute('create table if not exists osoba '
               '(imie, czy_mezczyzna, wiek)')
cursor.close()

# wstawianie nowych rekordów danych
prefix = 'insert into osoba values '
cursor = conn.cursor()
cursor.execute(prefix + "('Adam', 1, 30)")
cursor.execute(prefix + "('Anna', 0, 25)")
cursor.execute(prefix + "('Robert', 1, 19)")
conn.commit()
cursor.close()
```

Przykład cd.

Przykład

```
# wykonanie kwerendy
cursor = conn.cursor()
cursor.execute("select * from osoba where czy_mezczyzna = 1")

# odebranie i wyświetlenie danych
for (imie, plec, wiek) in cursor.fetchall():
    print(f'{imie}, '
          f'{"mężczyzna" if plec == 1 else "kobieta"}, '
          f'wiek: {wiek}')
cursor.close()

# usunięcie tabeli
cursor = conn.cursor()
cursor.execute('drop table osoba')
cursor.close()

conn.close()
```

Zagrożenia

- Uwaga: Należy **bezwzględnie walidować** zmienne generujące zapytania SQL, aby się ustrzec przed atakami typu *SQL Injection*

Przykładowe zapytanie

```
query = "SELECT * FROM salary WHERE name='{}'"
curs.execute(query.format(name))
```

- Przy poprawnej wartości parametru otrzymamy prawidłowy SQL:

Prawidłowe działanie

```
# dla parametru:
name = 'Jan Nowak'

# treść zapytania:
SELECT * FROM salary WHERE name='Jan Nowak';
```

Zagrożenia – ataki *SQL Injection*

Mamy problem...

```
# dla parametru:
name = "' or 1=1; SELECT * FROM passwords; --"

# treść zapytania:
SELECT * FROM salary WHERE name='' or 1=1;
SELECT * FROM passwords; --'
```

Mamy DUŻY problem...

```
# dla parametru:
name = "' or 1=1; DROP TABLE passwords; --"

# treść zapytania:
SELECT * FROM salary WHERE name='' or 1=1;
DROP TABLE passwords; --'
```

Zagrożenia

- Aby ustrzec się ataków typu *SQL Injection* nie należy parametrów ujmować w cudzysłowy lub apostrofy – lepiej ująć je w nawiasy:

Poprawne maskowanie

```
query = "SELECT * FROM salary WHERE name=?"
cur.execute(query, (name,))
```

Typy parametrów

- Parametry przekazywane do bazy danych za pomocą symboli zastępczych muszą zazwyczaj być właściwego typu: liczbowego, tekstowego lub *None* (aby reprezentować SQL NULL)
- Nie istnieje typ powszechnie używany do reprezentowania dat, czasu i dużych obiektów binarnych (BLOB)

Typy parametrów

- Moduł zgodny z DB API dostarcza funkcji fabryki do budowy takich obiektów

Metody fabryki

```
Binary(string)
Date(year, month, day)
DateFromTicks(s)
```

Binary	zwraca obiekt reprezentujący podany łańcuch bajtów jako BLOB
Date	zwraca obiekt reprezentujący podaną datę
DateFromTicks	zwraca obiekt reprezentujący datę po upływie s sekund od początku epoki (wg modułu *time*)

Typy parametrów

Metody fabryki

```
Time(hour, minute, second)
TimeFromTicks(s)
Timestamp(year, month, day, hour, minute, second)
TimestampFromTicks(s)
```

Time	zwraca obiekt reprezentujący podany czas
TimeFromTicks	zwraca obiekt reprezentujący czas po upływie *s* sekund od początku epoki (wg modułu *time*)
Timestamp	zwraca obiekt reprezentujący podaną datę i czas
TimestampFromTicks	zwraca obiekt reprezentujący datę i czas po upływie *s* sekund od początku epoki (wg modułu *time*)

Standard DB API 2.0

- Poza relacyjnymi bazami danych istnieje wiele baz NoSQL
- Wśród nich można wskazać m.in.:
  - obiektowe bazy danych, takie jak: ZODB, Dobbin
  - bazy dokumentów, np.: MongoDB

Moduł *pymongo*

- Wsparcia dla komunikacji z bazą MongoDB z poziomu Pythona dostarcza moduł *pymongo*
- Baza MongoDB to baza dokumentów
- Pełna dokumentacja modułu jest dostępna pod tym linkiem

Moduł *pymongo*

- Kolekcję dokumentów reprezentuje klasa *Collection*
- Klasa udostępnia wiele metod, w tym metody zliczające dokumenty w kolekcji, umożliwiające zmianę nazwy kolekcji oraz jej usunięcie

Kolekcja dokumentów

```
count_documents(filter, session=None, **kwargs)

rename(new_name, session=None, **kwargs)

drop(session=None)
```

Moduł *pymongo*

Kolekcja dokumentów

```
from pymongo import MongoClient

# utworzenie klienta MongoDB
mongo = MongoClient('mongodb://localhost:27017')

# wybór/utworzenie bazy danych
baza = mongo['magazyn']     # baza = mongo.magazyn

# wybór/utworzenie kolekcji dokumentów
klienci = baza['klienci']  # klienci = baza.klienci

# lista kolekcji dokumentów
kolekcje = baza.list_collection_names()

print('kolekcje =', kolekcje)
kolekcje = ['klienci']
```

Moduł *pymongo*

Metody zmieniające zawartość kolekcji dokumentów

```
insert_one(document, bypass_document_validation=False,
           session=None)
insert_many(documents, ordered=True,
            bypass_document_validation=False, session=None)

replace_one(filter, replacement, upsert=False,
            bypass_document_validation=False, collation=None,
            hint=None, session=None)

update_one(filter, update, upsert=False,
           bypass_document_validation=False, collation=None,
           array_filters=None, hint=None, session=None)
update_many(filter, update, upsert=False, array_filters=None,
            bypass_document_validation=False, collation=None,
            hint=None, session=None)

delete_one(filter, collation=None, hint=None, session=None)
delete_many(filter, collation=None, hint=None, session=None)
```

Moduł *pymongo*

Metody wyszukujące dokumenty

```
find(filter=None, projection=None, skip=0, limit=0, no_cursor_timeout=False,
     cursor_type=CursorType.NON_TAILABLE, sort=None, allow_partial_results=False,
     oplog_replay=False, modifiers=None, batch_size=0, manipulate=True,
     collation=None, hint=None, max_scan=None, max_time_ms=None, max=None, min=None,
     return_key=False, show_record_id=False, snapshot=False, comment=None,
     session=None)

find_one(filter=None, *args, **kwargs)

find_one_and_delete(filter, projection=None, sort=None, hint=None, session=None,
                    **kwargs)

find_one_and_replace(filter, replacement, projection=None, sort=None,
                     return_document=ReturnDocument.BEFORE, hint=None, session=None,
                     **kwargs)

find_one_and_update(filter, update, projection=None, sort=None,
                        return_document=ReturnDocument.BEFORE, array_filters=None,
                        hint=None, session=None, **kwargs)
```

Moduł *pymongo*

Przykład

```
klient = {'imie': 'Jan', 'nazwisko': 'Kowalski'}

# wstawienie dokumentu i odebranie jego id
id = klienci.insert_one(klient).inserted_id
print('id =', id)
id = 5fb812ef61e7c1ea241ad261

# wyszukanie dokumentu po id
klient = klienci.find_one({'_id': id})
print(klient)
klient = {'_id': ObjectId('5fb812ef61e7c1ea241ad261'),
          'imie': 'Jan', 'nazwisko': 'Kowalski'}

# wyszukanie dokumentu wg kryteriów
klient = klienci.find_one({'nazwisko': 'Kowalski'})
print(klient)
klient = {'_id': ObjectId('5fb7f520bb58bdc4149bf92e'),
          'imie': 'Jan', 'nazwisko': 'Kowalski'}
```

ORM

- **ORM** (*Object-Relational Mapping*) – to narzędzie mapowania obiektowo-relacyjnego
- Umożliwia komunikację z relacyjną bazą danych poprzez odwzorowanie modelu obiektowego, czyli klas encyjnych (model abstrakcyjny) na tabele w bazie danych (model fizyczny)
- W takim przypadku tabele są tworzone automatycznie przez usługę, podobnie jak zapytania SQL
- Istnieje kilka implementacji ORM dla Pythona, wśród nich:

MODUŁ	DOKUMENTACJA
peewee	☞ link
SQLAlchemy	☞ link

Biblioteka *SQLAlchemy*

- SQLAlchemy umożliwia stworzenie abstrakcji i uniezależnienie się od bazy danych i dialektu SQL który ona wykorzystuje
- Najpierw należy utworzyć silnik:

Utworzenie silnika

```
create_engine(*args, **kwargs)
```

- Funkcja wykorzystuje URL-a do bazy danych o postaci:
  dialect+driver://username:password@host:port/database[?key=value,...]
  gdzie:

dialect	np. *sqlite, mysql, postgresql, oracle, mssql*
driver	nazwa DB API, np.: *psycopg2, pyodbc, cx_oracle*

Biblioteka *SQLAlchemy*

- Opcjonalne parametry nazwane, to m.in.:

echo=False	logowanie poleceń na standardowe wyjście
encoding='utf8'	domyślne kodowanie znaków
convert_unicode=False	domyślne konwertowanie łańcuchów w bazie na Unicode
execution_options	np. *autocommit*
listeners	listenery na zdarzenia związane z pulą połączeń

Przykład

```
from sqlalchemy import create_engine

silnik = create_engine('mysql+pymysql://root:admin@localhost/'
                       'magazyn?charset=utf8mb4', echo=True)
```

Biblioteka *SQLAlchemy*

- Silnik posiada wiele metod:

Wybrane metody silnika

```
begin(close_with_result=False)
connect(**kwargs)
execute(statement, *multiparams, **params)
table_names(schema=None, connection=None)
transaction(callable_, *args, **kwargs)
```

begin	zwraca kontekst
connect	zwraca połączenie
execute	wykonuje polecenie
table_names	zwraca dostępne tabele w bazie
transaction	opakowuje funkcję w transakcję

Biblioteka *SQLAlchemy*

- Aby dokonać mapowania klas encyjnych na bazę danych potrzebujemy bazy deklaratywnej
- Będzie ona stanowiła nadklasę klas encyjnych
- Dalej pozostaje wygenerowanie schematu bazy danych (utworzenie tabel) i utworzenie instancji encji

Biblioteka *SQLAlchemy*

Klasa encyjna (moduł *encje*)

```
from sqlalchemy.ext.declarative import *

Baza = declarative_base()

# klasa encyjna
class Klient(Baza):
    __tablename__ = 'klienci'
    id = Column(Integer, primary_key=True)
    imie = Column(String(20))
    nazwisko = Column(String(30))
    wiek = Column(Integer)
    mezczyzna = Column(Boolean, default=True)

    def __init__(self, imie,nazwisko, wiek, mezczyzna=True):
        self.imie = imie
        self.nazwisko = nazwisko
        self.wiek= wiek
        self.mezczyzna = mezczyzna

    def __str__(self):
        return (f'{self.imie} {self.nazwisko}, '
                f'{"mężczyzna" if self.mezczyzna else "kobieta"}, '
                f'wiek: {self.wiek}')
```

Biblioteka *SQLAlchemy*

Instancje encji

```
from sqlalchemy import create_engine
from encje import Baza, Klient

silnik = create_engine('mysql+pymysql://root:admin@localhost/'
                       'magazyn?charset=utf8mb4', echo=False)

# utworzenie schematu bazy danych
Baza.metadata.create_all(silnik)

klient1 = Klient('Jan', 'Kowalski', 30)
klient2 = Klient('Anna', 'Nowakowska', 25, False)

print(klient1)
Jan Kowalski, mężczyzna, wiek: 30

print(klient2)
Anna Nowakowska, kobieta, wiek: 25
```

Biblioteka *SQLAlchemy*

- Wszystkie operacje na bazie danych są wykonywane poprzez obiekt sesji – reprezentuje go klasa *Session*

Tworzenie sesji

```
from sqlalchemy.orm import sessionmaker

Sesja = sessionmaker(bind = silnik)
sesja = Sesja()
```

Biblioteka *SQLAlchemy*

- Wybrane atrybuty funkcji *sessionmaker*:

bind	powiązanie sesji z konkretnym połączeniem
autoflush	natychmiastowe zmiany (ale nie zatwierdzone) zatwierdzenie na końcu po wywołaniu *commit*
autocommit	natychmiastowe zatwierdzanie zmian w odrębnych chwilowych transakcjach
expire_on_commit	wygasza sesję po zatwierdzeniu zmian

- Zanim stan obiektu zostanie utrwalony w bazie, musi być dodany do sesji

Biblioteka *SQLAlchemy*

- Wybrane metody sesji:

add	dodaje obiekt do sesji
add_all	dodaje kolekcję obiektów do sesji
begin	początek transakcji
begin_nested	początek transakcji zagnieżdżonej
commit	zatwierdzenie zmian
rollback	wycofanie się z zapisanych zmian
dirty	obiekty zmienione
new	obiekty nowe
execute	wykonanie polecenia
scalar	jak *execute*, ale z liczbą w wyniku
flush	zapis zmian, ale jeszcze nie zatwierdzony
query	zapytanie

Biblioteka *SQLAlchemy*

- Obiekt encyjny może znajdować się w jednym z poniższych stanów:

transient	instancja nie znajduje się w sesji i nie jest zapisana w bazie
pending	po wywołaniu metody *add*, jest w sesji, ale nie jest zapisana w bazie aż do wywołania metody *flush*
persistent	instancja jest w sesji i w bazie
detached	jest w bazie, ale nie ma jej w żadnej sesji

Biblioteka *SQLAlchemy*

Przykład

```
sesja.add_all([klient1, klient2])
sesja.commit()

klient = sesja.query(Klient) \
              .filter_by(nazwisko='Nowakowska') \
              .first()

print(f'{klient.id}: {klient}')
2: Anna Nowakowska, kobieta, wiek: 25
```

Dziękujemy za uwagę