Kurs rozszerzony języka Python

Wykład 10.

Marcin Młotkowski

10 grudnia 2024

Plan wykładu

Plan wykładu

Pakiet `pickle`

Pakiet implementujący *serializację* i *deserializację* obiektów.

[2] A właściwie zbiór formatów

Pakiet `pickle`

Pakiet implementujący *serializację* i *deserializację* obiektów.

Format[2] natywny Pythona.

[2] A właściwie zbiór formatów

Pakiet `pickle`

Pakiet implementujący *serializację* i *deserializację* obiektów.

Format[2] natywny Pythona.
Użyteczny do przechowywania pojedynczych obiektów.

[2] A właściwie zbiór formatów

Jak korzystać

```
import picke
obj1 = {"uno": [1], "duo": [2,3], "tres": [4,5,6]}
```

Zapis

```
with open("object.store", 'wb') as fh:
    pickle.dump(obj1, fh)
```

Odczyt

```
with open("object.store", 'rb') as fh:
    obj2 = pickle.load(fh)
```

```
print(obj2)
```

Co można przechowywać

- wartości proste (True, False, liczby);
- listy, stringi, krotki, słowniki;
- klasy, obiekty (spełniające pewne warunki), funkcje.

json

```
import json
```

używamy dokładnie tak samo jak `pickle`

Uwagi

- można używać kompresji;
- podatność na ataki (niezaufane pliki), można skorzystać z podpisywania, np. HMAC;
- jest sześć wariantów serializacji w zależności od wersji Pythona, można jawnie wskazywać której wersji się używa.

Pakiet `shelve`

Pakiet do przechowywania w pliku większej ilości obiektów w postaci słownika.

- kluczem zawsze jest string;
- wartością jest obiekt zserializowany *picklem*;
- korzysta z tzw. *dbm'ow*, narzędzi dostępnych w bibliotekach uniksowych.

Przykład

```
import shelve

with shelve.open("shelve") as db:
    for i in range(10):
        db[f"lista{i}"] = [1,2,3, i]
    db.sync()
    for k in db:
        print(f"{k}: {db[k]}")
```

Shelve: uwagi

- otwierając plik można wskazać, czy zapis ma być częsty (po zmianie/aktualizacji);
- nie ma wielodostępu;
- trzeba się pilnować:

```
db['lista'] = [1,2,3]
db['lista'].append(4)
```

nie zmienia listy w db, ale

```
tmp = db['lista']
tmp.append(4)
db['lista'] = tmp
```

działa dobrze.

Po co mi to

Przydało mi się:

Cache zapytań SQL

```
SELECT * FROM dz_transakcje WHERE status ... : 389 504
SELECT * FROM dz_programy                    :     868
select * from dz_kody_kasowe order by typ, i :     558
```

10-krotne przyspieszenie działania programu.

Plan wykładu

Silniki SQL

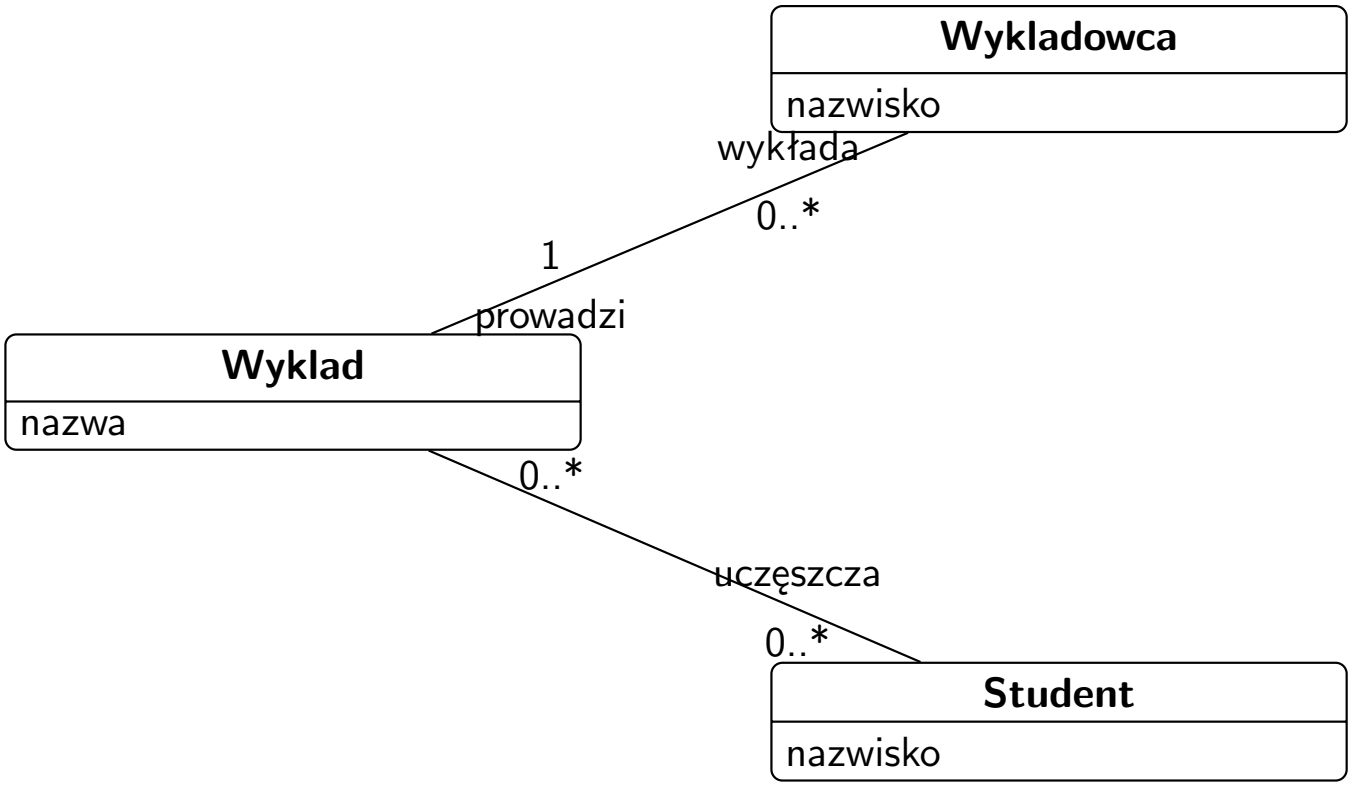
- Oracle
- DB/2
- MySQL
- PostgreSQL
- MSSQL
- ...

DB API

Python Database API Specification (PEP 249)

Zunifikowany interfejs dostępu do różnych systemów BD. Obecna wersja: 2.0.

Przykładowa baza danych



SQLite

- 'Plikowa' baza danych, bez zewnętrznego serwera, żadnego kontaktu z adminem;
- moduł: sqlite3
- Implementuje DB API 2.0 z rozszerzeniami

Otwarcie połączenia z serwerem BD

```
connect("parametry") # zwraca obiekt Connection
```

Otwarcie połączenia z serwerem BD

```
connect("parametry") # zwraca obiekt Connection
```

SQLite3

```
import sqlite3
db = sqlite3.connect("zapisy.db")
# można też: db = sqlite.connect(":memory:")
```

Zamknięcie połączenia

```
db.close()
```

Komunikacja z bd

Wysłanie zapytania

```
wynik = db.cursor()
wynik.execute("SELECT * FROM Studenci")
```

Komunikacja z bd

Wysłanie zapytania

```
wynik = db.cursor()
wynik.execute("SELECT * FROM Studenci")
```

pobranie wyniku

```
for st in wynik:
    print(w)
```

Opcjonalnie

```
wynik.close()
```

Wynik: obiekt klasy Cursor

Atrybuty wyniku:

- description: opisuje kolumny
- rowcount: liczba przetworzonych wierszy (np. INSERT czy UPDATE)
- ...

DB API: dodatkowe informacje

Standardowe wyjątki:
Warning, DatabaseError, NotSupportedError, ...

Użycie silnika MySQL

```
import MySQLdb

db = MySQLdb.connect(host="localhost",
                     db="nowa",
                     user="user",
                     passwd="123456")
```

Użycie silnika PostgreSQL

```
import psycopg2

db = psycopg2.connect(database="testowa",
        host="localhost",
        user="user",
        passwd="123456")
```

Użycie silnika Microsoft SQL Server

```
import pymssql

db = pymssql.connect("127.0.0.1", "user",
                     "haslo123456", "bazadanych")
```

Plan wykładu

Po co SQLAlchemy

To zajęcia Pythona a nie z SQL'a!

Po co SQLAlchemy

To zajęcia Pythona a nie z SQL'a!

Object–Relational Mapping (ORM)

Sposób odwzorowania świata obiektów w programie na świat relacyjny w bazie danych.

Co daje nam ORM

Używając tylko Pythona możemy

- utworzyć tabele w bazie danych;
- tworzyć, odczytywać, aktualizować i usuwać dane (CRUD: Create, Read, Update, Delete);
- definiować różne sposoby komunikacji danych (leniwość/gorliwość, transakcyjność, etc).

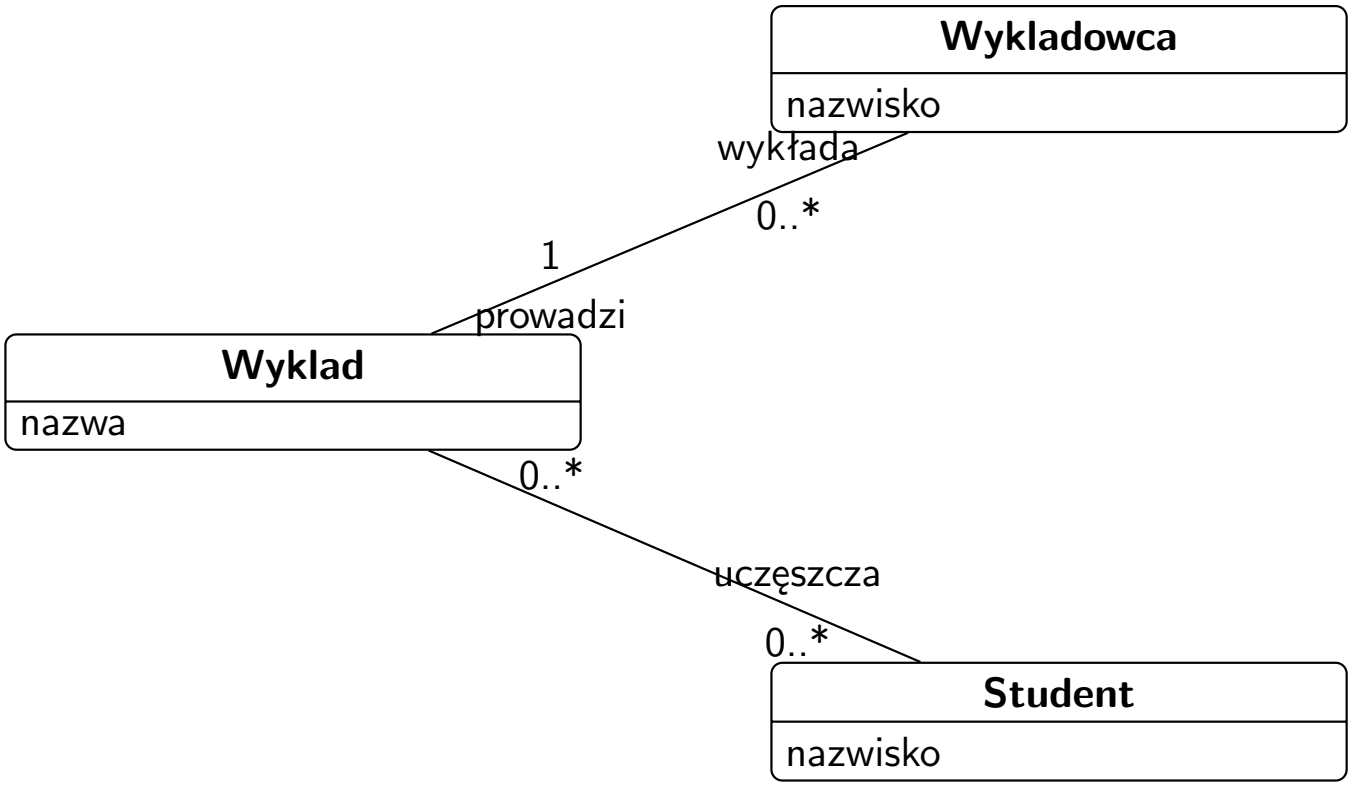
Potrzebne moduły

```
from sqlalchemy.orm import DeclarativeBase
from sqlalchemy import Table, Column, Integer, ForeignKey,
                       String
from sqlalchemy.orm import relationship, mapped_column,
                       Mapped

from __future__ import annotations
from typing import List

class Base(DeclarativeBase):
    pass
```

Przykładowa baza danych

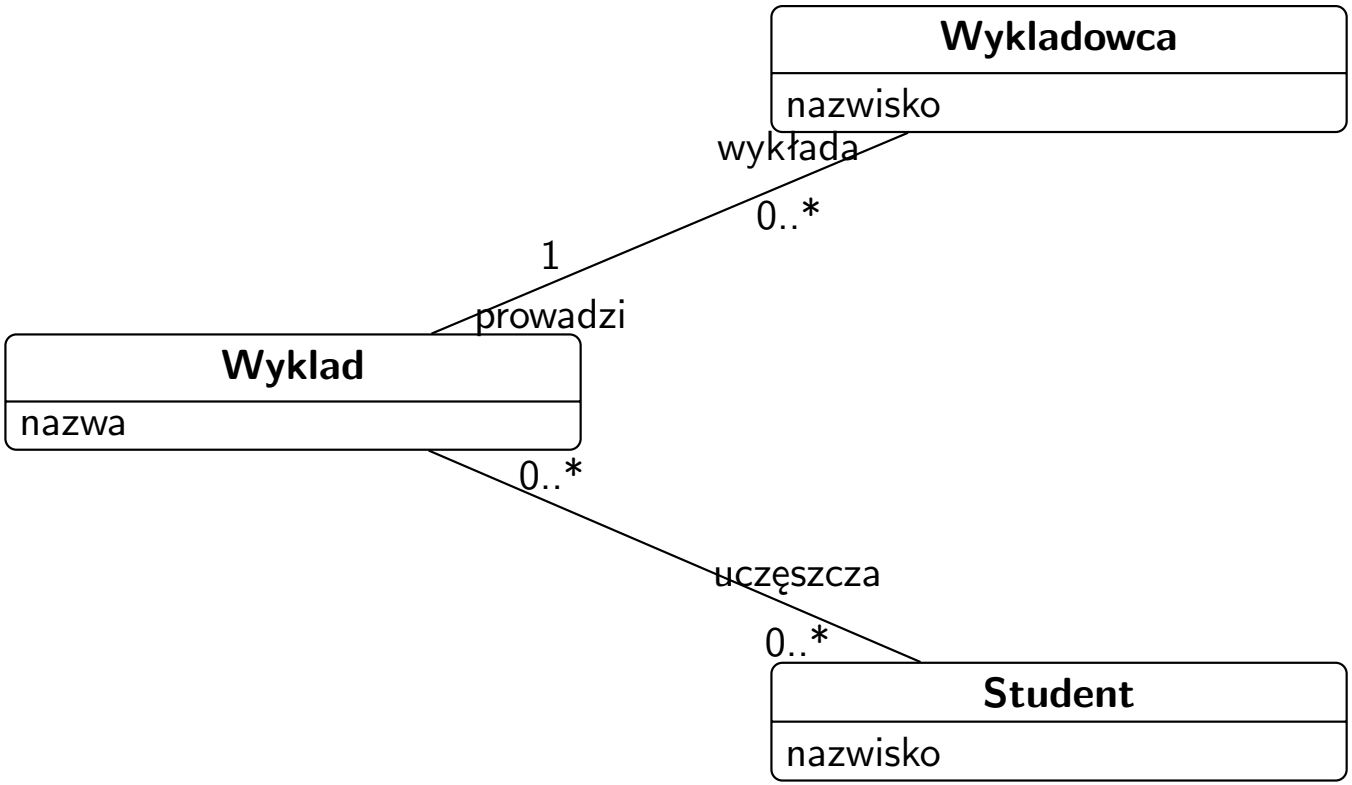


Deklaracja modeli, relacja one–to–many

```
class Wykladowca(Base):
    __tablename__ = "Wykladowcy"
    id = mapped_column(Integer, primary_key=True)
    nazwisko = mapped_column(String)
    wyklada: Mapped[List[Wyklad]] =
      relationship("Wyklad", back_populates="wykladowca")

class Wyklad(Base):
    __tablename__ = "Wyklady"
    id = mapped_column(Integer, primary_key=True)
    nazwa = mapped_column(String)
    ## Związek z Wykładowcą
    wykladowca_id =
        mapped_column(Integer,
            ForeignKey('Wykladowcy.id'))
    wykladowca =
        relationship("Wykladowca",
            back_populates="wyklada")
```

Przykładowa baza danych



Model Studenci, many–to–many

```
zapisy = Table(
    "student_wyklad",
    Base.metadata,
    Column("wyklad_id", ForeignKey("Wyklady.id")),
    Column("student_id", ForeignKey("Studenci.id"))

class Wyklad(Base):
    ...
    zapisani: Mapped[List[Student]] =
        relationship(secondary=zapisy)

class Student(Base):
    __tablename__ = 'Studenci'

    id = Column(Integer, primary_key=True)
    nazwisko = Column(String)
    zapisany: Mapped[List[Wyklad]] =
        relationship(secondary=zapisy)
```

Walidacje danych

```
from sqlalchemy.orm import validates

class Wykladowca(Base):
    ...
    @validates("nazwisko")
    def validate_nazwisko(self, key, nazwisko):
        if len(nazwisko) < 3:
            raise ValueError("Nazwisko za krótkie")
        return nazwisko
```

Utworzenie tabeli

```
from sqlalchemy import create_engine
engine = create_engine("sqlite:///wyklad.db", echo=True)
Base.metadata.create_all(engine)
```

Migracje

A co ze zmianą struktury bazy danych?

Migracje

A co ze zmianą struktury bazy danych?

alembic

CRUD

CRUD: Create, Read, Update, Delete

Sesja

Operacje odbywają się w ramach sesji:

```
from sqlalchemy.orm import sessionmaker

engine = create_engine("sqlite:///wyklad.db", echo=True)

with Session(engine) as session:
    wykladowca = Wykladowca(nazwa="Albert Einstein")
    wyklad = Wyklad(nazwa="Fizyka relatywistyczna",
        wykladowca=wykladowca.id)

    session.add(wykladowca)
    session.add(wyklad)

    session.commit()
```

Wiązanie danych

```
wyklad = Wyklad(nazwa="Analiza")

s1 = Student(nazwisko="Jan Nowak")
s2 = Student(nazwisko="Maksym Debeściak")

wyklad.zapisani.append(s1)
s1.append.zapisany(wyklad)

wyklad.zapisani.append(s2)
s2.append.zapisany(wyklad)

session.add_all([wyklad, s1, s2])
sesja.commit()
```

Wyszukiwanie (Read)

```
lista = sesja.query(Student).
          filter(Student.nazwisko.in_(["Debeściak"]).all()
```

Aktualizacja (Update)

```
student.nazwisko = "Nowy"
```

Usuwanie (Delete)

```
session.delete(wyklad)
```

Uwagi

- wycofywanie zmian: `session.rollback();`
- na końcu dobrze jest zrobić `sesja.close();`
- sesja nie jest dla wielu wątków;
- są dedykowane warianty typów kolumn i zapytań związanych ze specyfiką poszczególnych silników.

Dostęp asynchroniczny

```
from sqlalchemy.ext.asyncio import create_async_engine
from sqlalchemy.ext.asyncio import async_sessionmaker
```

Silnik asynchroniczny

```
async def async_main():
    engine = create_async_engine("sqlite+aiosqlite:///zapis
    async_session = async_sessionmaker(engine, expire_on_co

    await operacja1(async_session)
    await operacja2(async_session)

    await engine.dispose()

asyncio.run(async_main())
```

Sesje

```
async def operacja1(async_session):
    async with async_session() as session:
        async with session.begin():
            session.add_all([

                Wykladowca(nazwisko="Tarjan"),
                Wykladowca(nazwisko="Knuth"),
                Wykladowca(nazwisko="Pacholski")
            ]
            )
```

Plan wykładu

NoSQL

NOSQL (not only SQL)

Systemy baz danych o elastycznej strukturze danych. Czasem mówi się że są to ustrukturalizowane zasoby.

NoSQL

NOSQL (not only SQL)

Systemy baz danych o elastycznej strukturze danych. Czasem mówi się że są to ustrukturalizowane zasoby.

Do czego się używa

Proste, lecz wielkie bazy danych przetwarzane na wielu komputerach.

Systemy zorientowane na dokumenty

Dokument

Dokument zawiera jakąś informację. Dokument może być w formacie XML, YAML, JSON, PDF, MS Office. Dokumenty nie muszą mieć jednego schematu.

Systemy zorientowane na dokumenty

Dokument

Dokument zawiera jakąś informację. Dokument może być w formacie XML, YAML, JSON, PDF, MS Office. Dokumenty nie muszą mieć jednego schematu.

Przykłady danych

```
Imie="Adam"

Imie="Janina", Adres="ul. Cicha 132 m. 16",
    Dzieci=["Staś", "Krzyś" ]
```

Inne cechy

Dokumenty mają unikatowe klucze (string, URI).

Inne cechy

Dokumenty mają unikatowe klucze (string, URI).

Wyszukiwanie

Wyszukiwanie oparte na kluczu lub zawartości.

Przykłady systemów

CouchDB, MongoDB, Redis

MongoDB

System MongoDB

- system zorienowany na dokumenty;
- kolekcja: coś w rodzaju tabeli;
- nazwa modułu: `pymongo`;
- wymaga uruchomionego serwera `mongod`.

Połączenie z MongoDB

```
from pymongo import MongoClient
```

Włożenie danych

```
with MongoClient() as client:
    db = client['wyklad']

    osoby = db['osoby']

    adr1 = {'ulica': "Amphiteatre Parkway",
            'miasto': 'Mountain View'}
    adr2 = {'ulica':
            "One Microsoft Way", 'miasto': 'Redmond'}
    osoba1 = { "imie": 'Debeściak',
               'adresy': [ adr1, adr2 ] }

    osoby.insert_one(osoba1) # inserted.id
```

Pobranie danych

```
with MongoClient() as client:
    db = client['wyklad']

    wynik = db.osoby.find({'imie':
                { '$regex' : "/*.ebeś.*" }})
    for elem in wynik:
        print(elem)
```

Grafowe bazy danych

Dane są trzymane w postaci grafów: węzły reprezentują obiekty, a krawędzie: związki między obiektami.

Zastosowanie

Mapy, systemy geograficzne, dokumenty etc.