Dzień 1: Praca z bazą danych - Część 1

Data: 29.01.2026 | **Godziny:** 14:00-17:00

Przygotowanie środowiska

1. Wirtualne środowisko Python

Wirtualne środowisko izoluje zależności projektu od systemowego Pythona. Każdy projekt powinien mieć własne środowisko.

Tworzenie wirtualnego środowiska

```
# Przejdź do katalogu projektu
cd /sciezka/do/projektu

# Utwórz wirtualne środowisko (nazwa 'venv' jest konwencją)
python -m venv venv

# Alternatywnie z konkretną wersją Pythona
python3.11 -m venv venv
```

Aktywacja środowiska

```
# Linux / macOS
source venv/bin/activate

# Windows (Command Prompt)
venv\Scripts\activate.bat

# Windows (PowerShell)
venv\Scripts\Activate.ps1
```

Po aktywacji zobaczysz `(venv)` na początku linii w terminalu.

Dezaktywacja środowiska

```
deactivate
```

2. Instalacja pakietów

Podstawowe komendy pip

```
# Upewnij się, że środowisko jest aktywne!
(venv) $ pip install nazwa_pakietu

# Instalacja konkretnej wersji
pip install psycopg2-binary==2.9.9

# Instalacja wielu pakietów
pip install psycopg2-binary pymysql python-dotenv

# Instalacja z pliku requirements.txt
pip install -r requirements.txt

# Sprawdzenie zainstalowanych pakietów
pip list

# Zapisanie aktualnych zależności do pliku
pip freeze > requirements.txt
```

Przykładowy plik requirements.txt

```
psycopg2-binary==2.9.9
pymysql==1.1.0
python-dotenv==1.0.0
```

3. Szybka ścieżka - PyCharm (zalecane)

Jeśli używasz PyCharm, możesz sklonować repozytorium i skonfigurować środowisko bez użycia terminala.

Krok 1: Klonowanie repozytorium z GitHub

1. Uruchom PyCharm
2. Na ekranie powitalnym wybierz **Get from VCS** (lub **Clone Repository**)
    - Jeśli masz otwarty projekt: **Git → Clone...** lub **VCS → Get from Version Control**
3. Kliknij ikonę **GitHub** po lewej stronie
4. Kliknij **Log In via GitHub...** (otworzy się przeglądarka)
5. Zaloguj się na swoje konto GitHub i autoryzuj PyCharm
6. Po zalogowaniu zobaczysz listę swoich repozytoriów
7. Wybierz repozytorium z listy (np. `ergo-hestia-szkolenie`)
8. Wybierz katalog docelowy w polu **Directory**
9. Kliknij **Clone**

Krok 2: Konfiguracja wirtualnego środowiska w PyCharm

Po sklonowaniu PyCharm zapyta o interpreter Pythona:

1. Wybierz **Add New Interpreter → Add Local Interpreter...**

2. Wybierz **Virtualenv Environment**
3. Zaznacz **New** i upewnij się, że ścieżka to `venv` w katalogu projektu
4. Wybierz **Base interpreter** (np. Python 3.11)
5. Kliknij **OK**

Jeśli PyCharm nie zapytał automatycznie:

1. **File → Settings** (lub **PyCharm → Preferences** na macOS)
2. **Project: nazwa-projektu → Python Interpreter**
3. Kliknij ikonę ⚙️ → **Add...**
4. Postępuj jak wyżej

Krok 3: Instalacja pakietów w PyCharm

Metoda 1: Przez interfejs

1. **File → Settings → Project → Python Interpreter**
2. Kliknij **+** (Install packages)
3. Wyszukaj pakiet (np. `psycopg2-binary`)
4. Kliknij **Install Package**

Metoda 2: Przez requirements.txt (szybsze)

1. Otwórz plik `requirements.txt` w edytorze
2. PyCharm wyświetli żółty pasek z informacją o brakujących pakietach
3. Kliknij **Install requirements**

Metoda 3: Przez terminal PyCharm

1. Otwórz terminal: **View → Tool Windows → Terminal**
2. Terminal automatycznie aktywuje środowisko wirtualne projektu
3. Wpisz: `pip install -r requirements.txt`

Krok 4: Pobieranie zmian z GitHub (git pull)

1. **Git → Pull...** (lub skrót **Ctrl+T** / **Cmd+T**)
2. Upewnij się, że wybrana jest właściwa gałąź (np. `origin/main`)
3. Kliknij **Pull**

Lub użyj niebieskiej strzałki ↓ na pasku narzędzi.

Krok 5: Wysyłanie zmian na GitHub (git push)

1. **Git → Commit...** (lub **Ctrl+K** / **Cmd+K**)
2. Zaznacz pliki do wysłania
3. Wpisz opis commita
4. Kliknij **Commit and Push...** (lub najpierw **Commit**, potem **Git → Push**)

---

4. Konfiguracja Git i GitHub (terminal)

Alternatywna ścieżka dla osób preferujących terminal.

Instalacja Git

```
# Linux (Debian/Ubuntu)
sudo apt install git

# macOS (z Homebrew)
brew install git

# Windows – pobierz z https://git-scm.com/downloads
```

Konfiguracja Git (jednorazowo)

```
# Ustaw swoją tożsamość
git config --global user.name "Twoje Imię Nazwisko"
git config --global user.email "twoj.email@example.com"

# Sprawdź konfigurację
git config --list
```

Klonowanie repozytorium z GitHub

```
# Klonowanie przez HTTPS (wymaga loginu/tokena)
git clone https://github.com/uzytkownik/nazwa-repo.git

# Klonowanie przez SSH (wymaga klucza SSH)
git clone git@github.com:uzytkownik/nazwa-repo.git

# Klonowanie do konkretnego katalogu
git clone https://github.com/uzytkownik/nazwa-repo.git moj-katalog
```

Pobieranie zmian z repozytorium

```
# Pobierz i scal zmiany z zdalnego repozytorium
git pull

# Tylko pobierz zmiany (bez scalania)
git fetch

# Pobierz zmiany z konkretnej gałęzi
git pull origin main
```

Podstawowe operacje Git

```
# Sprawdź status zmian
git status

# Dodaj pliki do staging area
git add nazwa_pliku.py
git add .  # wszystkie zmienione pliki

# Zatwierdź zmiany (commit)
git commit -m "Opis zmian"

# Wyślij zmiany na GitHub
git push

# Wyślij nową gałąź
git push -u origin nazwa-galezi
```

Uwierzytelnianie na GitHub

Opcja 1: Token osobisty (HTTPS)

1. Wejdź na GitHub → Settings → Developer settings → Personal access tokens
2. Wygeneruj nowy token z uprawnieniami `repo`
3. Użyj tokena zamiast hasła przy `git push`

Opcja 2: Klucz SSH (zalecane)

```
# Wygeneruj klucz SSH
ssh-keygen -t ed25519 -C "twoj.email@example.com"

# Uruchom agenta SSH
eval "$(ssh-agent -s)"

# Dodaj klucz do agenta
ssh-add ~/.ssh/id_ed25519

# Skopiuj klucz publiczny
cat ~/.ssh/id_ed25519.pub
# Wklej go na GitHub: Settings → SSH and GPG keys → New SSH key

# Przetestuj połączenie
ssh -T git@github.com
```

4. Szybki start projektu

```
# 1. Sklonuj repozytorium
git clone https://github.com/firma/ergo-hestia-szkolenie.git
cd ergo-hestia-szkolenie

# 2. Utwórz i aktywuj środowisko wirtualne
python -m venv venv
source venv/bin/activate  # Linux/macOS
# lub: venv\Scripts\activate  # Windows

# 3. Zainstaluj zależności
pip install -r requirements.txt

# 4. Zweryfikuj instalację
python -c "import sqlite3; print('SQLite OK')"
python -c "import psycopg2; print('psycopg2 OK')"
```

Cele edukacyjne

Po ukończeniu tego modułu uczestnik będzie potrafił:

- [ ] Wyjaśnić wzorzec DB-API 2.0 i jego znaczenie dla przenośności kodu
- [ ] Nawiązać połączenie z bazą SQLite, PostgreSQL i MySQL z poziomu Pythona
- [ ] Wykonywać zapytania SELECT z prawidłową parametryzacją
- [ ] Stosować metody `fetchone()`, `fetchall()`, `fetchmany()` w odpowiednich sytuacjach
- [ ] Używać context managera do bezpiecznego zarządzania połączeniami
- [ ] Konwertować wyniki zapytań na słowniki za pomocą row factory

---

Wprowadzenie teoretyczne

Dlaczego Python i bazy danych?

W branży ubezpieczeniowej dane to fundament działalności. Każda polisa, roszczenie czy klient to rekord w bazie danych. Python stał się standardem w przetwarzaniu i analizie tych danych ze względu na:

- Prostą składnię i szybki development
- Bogaty ekosystem bibliotek (Pandas, SQLAlchemy)
- Możliwość integracji z systemami legacy i nowoczesnymi rozwiązaniami

Architektura komunikacji Python-Baza danych

```
flowchart LR
    A[Aplikacja Python] --> B[Sterownik DB-API 2.0]
    B --> C[Protokół sieciowy]
    C --> D[(Baza danych)]

    subgraph Sterowniki
        E[sqlite3]
```

```
        F[psycopg2]
        G[pymysql]
        H[oracledb]
    end

    B -.-> E
    B -.-> F
    B -.-> G
    B -.-> H
```

Wzorzec DB-API 2.0 (PEP 249)

DB-API 2.0 to standard definiujący interfejs między Pythonem a bazami danych. Dzięki niemu:

- **Przenośność kodu** - ten sam kod działa z różnymi bazami po zmianie sterownika
- **Spójność API** - jednolite metody (`connect`, `cursor`, `execute`, `fetch*`)
- **Obsługa transakcji** - standardowe `commit()` i `rollback()`

Główne komponenty

Komponent	Rola	Przykład
Connection	Zarządza połączeniem z bazą	`conn = sqlite3.connect('db.sqlite')`
Cursor	Wykonuje zapytania i przechowuje wyniki	`cursor = conn.cursor()`
Exceptions	Hierarchia wyjątków dla obsługi błędów	`sqlite3.DatabaseError`

Biblioteki i narzędzia

Porównanie sterowników

Sterownik	Baza danych	Placeholder	Instalacja	Uwagi
`sqlite3`	SQLite	`?` lub `:name`	Wbudowany	Idealne do prototypowania
`psycopg2`	PostgreSQL	`%s` lub `%(name)s`	`pip install psycopg2-binary`	Najpopularniejszy dla PostgreSQL
`pymysql`	MySQL/MariaDB	`%s` lub `%(name)s`	`pip install pymysql`	Pure Python
`oracledb`	Oracle	`:name` lub `:1`	`pip install oracledb`	Następca cx_Oracle

Diagram wyboru sterownika

```
flowchart TD
    A[Wybór bazy danych] --> B{Typ bazy}
    B -->|Lokalna/plik| C[SQLite]
    B -->|Enterprise| D{Dostawca}
    D -->|PostgreSQL| E[psycopg2]
    D -->|MySQL/MariaDB| F[pymysql]
    D -->|Oracle| G[oracledb]
    C --> H[sqlite3 - wbudowany]
    E --> I[pip install psycopg2-binary]
    F --> J[pip install pymysql]
    G --> K[pip install oracledb]
```

Przykłady kodu

Podstawowe połączenie z SQLite

```
import sqlite3

# Sposób 1: Klasyczny (wymaga ręcznego zamknięcia)
conn = sqlite3.connect('baza.db')
cursor = conn.cursor()
cursor.execute("SELECT * FROM klienci")
wyniki = cursor.fetchall()
conn.close()

# Sposób 2: Context Manager (zalecany)
with sqlite3.connect('baza.db') as conn:
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM klienci")
    wyniki = cursor.fetchall()
# Połączenie zamyka się automatycznie
```

Połączenie z PostgreSQL

```
import psycopg2

# Parametry połączenia
params = {
    'host': 'localhost',
    'database': 'ergo_hestia',
    'user': 'app_user',
    'password': 'haslo123',
    'port': 5432
}

# Połączenie z context managerem
with psycopg2.connect(**params) as conn:
    with conn.cursor() as cursor:
```

```
        cursor.execute("SELECT * FROM polisy WHERE typ = %s", ('OC',))
        polisy = cursor.fetchall()
```

Parametryzacja zapytań

```
# NIGDY tak nie rób - podatne na SQL Injection!
nazwa = "'; DROP TABLE polisy; --"
cursor.execute(f"SELECT * FROM klienci WHERE nazwisko = '{nazwa}'")

# ZAWSZE używaj parametryzacji
# SQLite - placeholder ?
cursor.execute(
    "SELECT * FROM klienci WHERE nazwisko = ? AND miasto = ?",
    ('Kowalski', 'Warszawa')
)

# SQLite - nazwane parametry
cursor.execute(
    "SELECT * FROM klienci WHERE nazwisko = :nazwisko AND miasto =
:miasto",
    {'nazwisko': 'Kowalski', 'miasto': 'Warszawa'}
)

# PostgreSQL/MySQL - placeholder %s
cursor.execute(
    "SELECT * FROM klienci WHERE nazwisko = %s AND miasto = %s",
    ('Kowalski', 'Warszawa')
)
```

Metody pobierania wyników

```
cursor.execute("SELECT * FROM polisy WHERE typ = 'OC'")

# fetchone() - jeden rekord lub None
pierwszy = cursor.fetchone()
if pierwszy:
    print(f"Pierwsza polisa: {pierwszy}")

# fetchall() - wszystkie rekordy jako lista
wszystkie = cursor.fetchall()
print(f"Znaleziono {len(wszystkie)} polis")

# fetchmany(n) - n rekordów
paczka = cursor.fetchmany(10)
while paczka:
    for rekord in paczka:
        print(rekord)
    paczka = cursor.fetchmany(10)
```

```
# Iteracja po kursorze - najbardziej wydajna dla dużych zbiorów
cursor.execute("SELECT * FROM polisy")
for rekord in cursor:
    print(rekord)
```

Row Factory - wyniki jako słowniki

```
import sqlite3

conn = sqlite3.connect('baza.db')
conn.row_factory = sqlite3.Row  # Włącz tryb słownikowy

cursor = conn.cursor()
cursor.execute("SELECT * FROM klienci WHERE id = ?", (1,))

klient = cursor.fetchone()

# Dostęp jak do słownika
print(klient['imie'])
print(klient['nazwisko'])
print(klient['email'])

# Można też iterować po kluczach
print(klient.keys())  # ['id', 'imie', 'nazwisko', 'email', ...]

# Konwersja do prawdziwego słownika
klient_dict = dict(klient)
```

Dynamiczne budowanie zapytań

```
def wyszukaj_polisy(typ=None, min_wartosc=None, max_wartosc=None,
klient_id=None):
    """
    Wyszukuje polisy według dynamicznych kryteriów.
    Bezpieczne budowanie zapytań z parametryzacją.
    """
    zapytanie = "SELECT * FROM polisy WHERE 1=1"
    parametry = []

    if typ:
        zapytanie += " AND typ = ?"
        parametry.append(typ)

    if min_wartosc is not None:
        zapytanie += " AND wartosc >= ?"
        parametry.append(min_wartosc)

    if max_wartosc is not None:
        zapytanie += " AND wartosc <= ?"
```

```
        parametry.append(max_wartosc)

    if klient_id:
        zapytanie += " AND klient_id = ?"
        parametry.append(klient_id)

    with sqlite3.connect('baza.db') as conn:
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()
        cursor.execute(zapytanie, parametry)
        return [dict(row) for row in cursor.fetchall()]
```

Częste błędy i antywzorce

1. SQL Injection

```
# ZŁE - podatne na atak
def znajdz_klienta_zle(nazwisko):
    cursor.execute(f"SELECT * FROM klienci WHERE nazwisko = '{nazwisko}'")

# DOBRE - bezpieczne
def znajdz_klienta_dobrze(nazwisko):
    cursor.execute("SELECT * FROM klienci WHERE nazwisko = ?",
(nazwisko,))
```

Dlaczego to ważne? W systemie ubezpieczeniowym wyciek danych klientów (PESEL, adresy, historia szkód) to naruszenie RODO i potencjalne wielomilionowe kary.

2. Niezamykanie połączeń

```
# ZŁE - wyciek połączeń
def pobierz_dane_zle():
    conn = sqlite3.connect('baza.db')
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM klienci")
    return cursor.fetchall()
    # conn nigdy nie zostanie zamknięte!

# DOBRE - context manager
def pobierz_dane_dobrze():
    with sqlite3.connect('baza.db') as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM klienci")
        return cursor.fetchall()
```

3. Ładowanie wszystkiego do pamięci

```
# ZŁE - przy 10 mln rekordów zabraknie pamięci
cursor.execute("SELECT * FROM historia_szkod")
wszystkie = cursor.fetchall()  # OutOfMemoryError!

# DOBRE - przetwarzanie w paczkach
cursor.execute("SELECT * FROM historia_szkod")
while True:
    paczka = cursor.fetchmany(1000)
    if not paczka:
        break
    for rekord in paczka:
        przetwarzaj(rekord)

# LEPSZE - iteracja po kursorze
cursor.execute("SELECT * FROM historia_szkod")
for rekord in cursor:
    przetwarzaj(rekord)
```

4. Brak obsługi błędów

```
# ZŁE - brak obsługi wyjątków
def zapisz_polise(dane):
    conn = sqlite3.connect('baza.db')
    cursor = conn.cursor()
    cursor.execute("INSERT INTO polisy VALUES (...)", dane)
    conn.commit()

# DOBRE - pełna obsługa błędów
def zapisz_polise(dane):
    try:
        with sqlite3.connect('baza.db') as conn:
            cursor = conn.cursor()
            cursor.execute("INSERT INTO polisy VALUES (...)", dane)
            conn.commit()
            return True
    except sqlite3.IntegrityError:
        # Np. duplikat klucza
        return False
    except sqlite3.DatabaseError as e:
        # Logowanie błędu
        logger.error(f"Błąd bazy danych: {e}")
        raise
```

5. Hardkodowanie connection stringów

```
# ZŁE - dane wrażliwe w kodzie
conn = psycopg2.connect(
    host="prod-server.company.com",
```

```
    password="SuperTajneHaslo123!"
)

# DOBRE - konfiguracja z zewnątrz
import os
from dotenv import load_dotenv

load_dotenv()

conn = psycopg2.connect(
    host=os.getenv('DB_HOST'),
    database=os.getenv('DB_NAME'),
    user=os.getenv('DB_USER'),
    password=os.getenv('DB_PASSWORD')
)
```

Przypadki użycia w ubezpieczeniach

Scenariusz 1: Raportowanie dzienne polis

Codziennie o 6:00 system generuje raport aktywnych polis wygasających w ciągu 30 dni.

```
import sqlite3
from datetime import datetime, timedelta

def pobierz_wygasajace_polisy(dni_do_wygasniecia=30):
    """
    Pobiera polisy wygasające w określonym czasie.
    Używane do wysyłki przypomnień o odnowieniu.
    """
    data_graniczna = datetime.now() + timedelta(days=dni_do_wygasniecia)

    with sqlite3.connect('ergo_hestia.db') as conn:
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        cursor.execute("""
            SELECT p.numer, p.typ, p.data_do, k.imie, k.nazwisko, k.email
            FROM polisy p
            JOIN klienci k ON p.klient_id = k.id
            WHERE p.data_do <= ? AND p.data_do >= date('now')
            ORDER BY p.data_do
        """, (data_graniczna.strftime('%Y-%m-%d'),))

        return [dict(row) for row in cursor.fetchall()]
```

Scenariusz 2: Weryfikacja klienta przy likwidacji szkody

Podczas zgłaszania szkody system musi zweryfikować, czy klient ma aktywną polisę danego typu.

```python
def weryfikuj_polise(numer_polisy, pesel_klienta):
    """
    Weryfikuje ważność polisy przed rozpoczęciem likwidacji szkody.

    Returns:
        dict z informacją o polisie lub None jeśli nieważna
    """
    with sqlite3.connect('ergo_hestia.db') as conn:
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        cursor.execute("""
            SELECT p.*, k.imie, k.nazwisko
            FROM polisy p
            JOIN klienci k ON p.klient_id = k.id
            WHERE p.numer = ?
              AND k.pesel = ?
              AND p.data_od <= date('now')
              AND p.data_do >= date('now')
        """, (numer_polisy, pesel_klienta))

        wynik = cursor.fetchone()
        return dict(wynik) if wynik else None
```

Scenariusz 3: Batch processing - aktualizacja składek

Raz w roku system przelicza składki na podstawie nowych tabel aktuarialnych.

```python
def aktualizuj_skladki_batch(wspolczynnik_korekty, typ_polisy):
    """
    Aktualizuje składki dla wszystkich polis danego typu.
    Operacja batch - wykonywana poza godzinami pracy.
    """
    with sqlite3.connect('ergo_hestia.db') as conn:
        cursor = conn.cursor()

        # Najpierw pobierz liczbe rekordow do aktualizacji
        cursor.execute(
            "SELECT COUNT(*) FROM polisy WHERE typ = ?",
            (typ_polisy,)
        )
        liczba_rekordow = cursor.fetchone()[0]

        # Wykonaj aktualizację
        cursor.execute("""
            UPDATE polisy
            SET wartosc = wartosc * ?
            WHERE typ = ?
        """, (wspolczynnik_korekty, typ_polisy))
```

```
        conn.commit()
        return liczba_rekordow
```

Podsumowanie dnia

Kluczowe wnioski

1. **DB-API 2.0** to standard gwarantujący przenośność kodu między bazami danych
2. **Context manager** (`with`) to preferowany sposób zarządzania połączeniami
3. **Parametryzacja** zapytań jest obowiązkowa - chroni przed SQL Injection
4. **Row factory** ułatwia pracę z wynikami (dostęp po nazwie kolumny)
5. **Iteracja po kursorze** jest wydajniejsza niż `fetchall()` dla dużych zbiorów

Wzorzec do zapamiętania

```python
import sqlite3

def bezpieczne_zapytanie(sql, parametry=None):
    with sqlite3.connect('baza.db') as conn:
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()
        cursor.execute(sql, parametry or ())
        return [dict(row) for row in cursor.fetchall()]
```

Przygotowanie do następnych zajęć

Do zrobienia przed Dniem 2:

1. Zainstaluj `psycopg2-binary`: `pip install psycopg2-binary`
2. Przejrzyj dokumentację: https://docs.python.org/3/library/sqlite3.html
3. Wykonaj pracę domową

Czym zajmiemy się na Dniu 2:

- Transakcje (`commit`, `rollback`)
- Operacje INSERT, UPDATE, DELETE
- SQLAlchemy ORM - wprowadzenie

Materiały dodatkowe

Linki

- PEP 249 - DB-API 2.0
- Dokumentacja sqlite3
- Dokumentacja psycopg2

Cheat Sheet

```
POŁĄCZENIE
----------
conn = sqlite3.connect('baza.db')
conn = psycopg2.connect(host, database, user, password)

KURSOR
------
cursor = conn.cursor()
cursor.execute(sql)
cursor.execute(sql, params)

POBIERANIE WYNIKÓW
------------------
cursor.fetchone()      # jeden rekord
cursor.fetchall()      # wszystkie
cursor.fetchmany(n)    # n rekordów
for row in cursor:     # iteracja

PARAMETRYZACJA
--------------
SQLite:     ? lub :nazwa
PostgreSQL: %s lub %(nazwa)s

ROW FACTORY
-----------
conn.row_factory = sqlite3.Row
dict(row)  # konwersja na słownik
```