Zaawansowane Pliki i Dane Ustrukturyzowane

Gdzie jesteśmy?

- Opanowaliśmy fundamenty: bezpieczny odczyt/zapis plików i obsługę błędów. Nasz kod jest odporny.

Nowy problem:

- Praca ze ścieżkami jako stringami jest niewygodna i podatna na błędy.
- Jak zapisać w pliku cały obiekt, a nie tylko tekst?
- Jak w ustrukturyzowany sposób pracować z danymi tabelarycznymi (CSV) i zagnieżdżonymi (JSON)?

Cel na dziś: Nauczyć nasze programy rozumieć dane, z którymi pracują, i operować na nich w sposób nowoczesny i profesjonalny.

Plan Działania:

- Zadanie 1: Obiektowe Ścieżki (pathlib)
- Zadanie 2: Serializacja Obiektów (pickle)
- Zadanie 3: Czytanie Plików CSV (csv.DictReader)
- Zadanie 4: Zapisywanie Plików CSV (csv.DictWriter)
- Zadanie 5: Czytanie i Parsowanie JSON (json.load)
- Zadanie 6: Zapisywanie do JSON (json.dump)
- Zadanie 7: Walidacja Danych z Pydantic
- Zadanie 8: Pliki w Pamięci (io.StringIO)
- Zadanie 9: Profesjonalne Logowanie (logging)
- Podsumowanie.

Zadanie 1 - Obiektowe Ścieżki (pathlib)

Problem: Manipulowanie ścieżkami jako stringami jest niewygodne i podatne na błędy.

```
# Stary, zły sposób
import os
folder = "raporty"
plik = "raport_2023.txt"
sciezka = folder + os.sep + plik # os.sep? a uprawnienia?
```

Rozwiązanie: Moduł **pathlib** - traktuje ścieżki jako obiekty z metodami, a nie zwykłe stringi.

Zadanie: Stwórz skrypt, który generuje plik z raportem w odpowiedniej strukturze folderów.

a. Zaimportuj klasę **Path** z modułu pathlib oraz moduł datetime.

b. Stwórz obiekt **Path** wskazujący na folder **raporty_dzienne**.

c. Użyj operatora **/**, aby stworzyć ścieżkę do pliku **raport.txt** wewnątrz podfolderu o nazwie **RRRR-MM-DD** (użyj **datetime.date.today()** do uzyskania dzisiejszej daty).

d. Zanim zapiszesz plik, upewnij się, że cała struktura folderów istnieje. Użyj metody **.parent.mkdir(parents=True, exist_ok=True)** na obiekcie ścieżki do pliku.

e. Zapisz do pliku dowolny tekst (np. "To jest treść raportu.") używając wygodnej metody **.write_text().** Pamiętaj o podaniu **encoding="utf-8".**

f. Wczytaj zawartość pliku z powrotem za pomocą **.read_text()** i wyświetl ją, aby potwierdzić poprawność zapisu.

g. Na koniec, użyj atrybutów obiektu **Path**, aby wyświetlić:

- Pełną, absolutną ścieżkę do pliku **(.resolve()).**
- Nazwę pliku **(.name).**
- Folder nadrzędny **(.parent**).

Omówienie Zadania 1 - Obiektowe Ścieżki (pathlib)

Cel: Analiza rozwiązania i zrozumienie, jak pathlib upraszcza i uelastycznia kod.

Kluczowe korzyści:

- **Czytelność: folder / plik** jest o wiele czystsze niż **os.path.join**.
- **Bezpieczeństwo:** Odporność na błędy związane z separatorami (**/** vs \).
- **Wygoda:** Metody **.mkdir(), .write_text(), .read_text()** wykonują za nas całą pracę.

Zadanie 2 - Serializacja Obiektów (pickle)

Problem: Jak zapisać do pliku cały, złożony obiekt (np. instancję klasy, listę słowników), a nie tylko tekst?

Rozwiązanie: Serializacja - proces zamiany obiektu na strumień bajtów. W Pythonie służy do tego moduł pickle.

Narzędzia:

- **pickle.dump(obiekt, plik):** Serializuje obiekt i zapisuje go do pliku.
- **pickle.load(plik):** Odczytuje obiekt z pliku.

WAŻNE: Pliki **pickle** muszą być otwierane w trybie binarnym (**'wb', 'rb**')!

Zadanie: Zapisz i odczytaj stan gry.

a. Zaimportuj moduł **pickle**.

b. Stwórz prostą klasę **StanGry** z atrybutami, np. **nazwa_gracza**: str, **punkty**: int, **ekwipunek**: list. Dodaj metodę **__repr__** dla ładnego wyświetlania.

c. Stwórz instancję tej klasy, wypełniając ją przykładowymi danymi.

d. Użyj **with open("stan_gry.pkl", "wb") as f**: do otwarcia pliku binarnego do zapisu (**.pkl** to konwencja).

e. Wewnątrz bloku **with**, użyj **pickle.dump(twoj_obiekt, f),** aby zapisać obiekt do pliku.

f. W drugiej części skryptu, użyj **with open("stan_gry.pkl", "rb") as f:** do otwarcia pliku binarnego do odczytu.

g. Wewnątrz bloku, użyj **wczytany_stan = pickle.load(f),** aby odtworzyć obiekt.

h. Wyświetl wczytany obiekt i sprawdź jego typ **(type(wczytany_stan)),** aby potwierdzić, że odzyskałeś pełnoprawny obiekt klasy **StanGry**.

i. **Ostrzeżenie: pickle** jest potężny, ale **nie jest bezpieczny**. Nigdy nie "odpieklowuj" danych z niezaufanego źródła (np. pobranych z internetu).

Omówienie Zadania 2 - Serializacja Obiektów (pickle)

Cel: Analiza rozwiązania i zrozumienie, jak pickle "zamraża" i "odmraża" obiekty.

Kluczowe mechanizmy:

- **pickle** serializuje obiekt do strumienia bajtów.
- Wymaga trybu binarnego: **'wb'** (write binary) i **'rb'** (read binary).
- **pickle.dump(obiekt, plik)** zapisuje, **pickle.load(plik)** odtwarza.
- Odtworzony obiekt jest pełnoprawną instancją oryginalnej klasy.

Zadanie 3: Czytanie Plików CSV (csv.DictReader)

Problem: Dostęp do danych przez indeks (**wiersz[2]**) jest nieczytelny i kruchy. Zmiana kolejności kolumn w pliku psuje kod.

Lepsze rozwiązanie: csv.DictReader - czyta wiersze jako słowniki, używając nagłówka jako kluczy.

Mini-pokaz (jak to działa):

```
import csv
# WAŻNE: plik otwieramy z newline=''
with open("plik.csv", "r", newline="", encoding="utf-8") as f:
    czytnik = csv.DictReader(f)
    for wiersz in czytnik:
        # wiersz to teraz słownik! np. {'imie': 'Jan', 'pensja': '8000'}
        print(f"Pracownik {wiersz['imie']} zarabia {wiersz['pensja']}.")
```

Zadanie: Na podstawie powyższego wzorca, napisz funkcję **wczytaj_pracownikow(sciezka)**

a. **Przygotuj plik:** Stwórz plik **pracownicy.csv** z treścią:

```
imie,stanowisko,pensja
Jan,Programista,8000
Anna,Manager,12500
Piotr,Tester,nie_liczba
```

b. **Napisz funkcję: wczytaj_pracownikow(sciezka_pliku: str) -> list.**

c. W funkcji, użyj bloku **try...except FileNotFoundError.**

d. W bloku **try**, otwórz plik z **with** i stwórz **csv.DictReader.**

e. Iteruj po czytniku. Dla każdego wiersza (słownika):

- Użyj wewnętrznego bloku **try...except ValueError,** aby spróbować przekonwertować pensję na **int**.
- Jeśli się uda, dodaj przetworzony słownik do listy wyników.
- Jeśli się nie uda, zignoruj ten wiersz (lub wyświetl ostrzeżenie).

f. Funkcja ma zwrócić listę poprawnie wczytanych słowników.

Omówienie Zadania 3 - Czytanie Plików CSV (csv.DictReader)

Cel: Analiza rozwiązania i zrozumienie, jak zbudować solidny, odporny na błędy parser CSV.

Kluczowe elementy solidnego kodu:

- Odporność na brak pliku: Zewnętrzny blok try...except FileNotFoundError.
- Odporność na błędy formatu: Wewnętrzny blok try...except ValueError dla każdej linii.
- Czytelność: Użycie DictReader pozwala na dostęp do danych po nazwie (wiersz['pensja']) zamiast po indeksie.

Zadanie 4 - Zapisywanie Plików CSV (csv.DictWriter)

Problem: Jak zapisać listę słowników do pliku CSV, poprawnie obsługując separatory i cudzysłowy?

Rozwiązanie: csv.DictWriter - zapisuje słowniki jako wiersze, dbając o formatowanie.

Mini-pokaz (jak to działa):

```
import csv

dane_do_zapisu = [
    {'imie': 'Kasia', 'pensja': 9000},
    {'imie': 'Tomek', 'pensja': 11000}
]
# 1. Definiujemy nazwy kolumn (kolejność ma znaczenie!)
pola = ['imie', 'pensja']

with open("wynik.csv", "w", newline="", encoding="utf-8") as f:
    # 2. Tworzymy obiekt DictWriter
    writer = csv.DictWriter(f, fieldnames=pola)
    # 3. Zapisujemy nagłówek
    writer.writeheader()
    # 4. Zapisujemy wszystkie wiersze
    writer.writerows(dane_do_zapisu)
```

Zadanie: Na podstawie powyższego wzorca, napisz funkcję **zapisz_raport_sprzedazy(sciezka, dane_sprzedazy).**

a. **Przygotuj dane:** Stwórz listę słowników **sprzedaz,** gdzie każdy słownik ma klucze **"produkt", "sprzedana_ilosc" i "przychody".**

b. **Napisz funkcję: zapisz_raport_sprzedazy(sciezka_pliku: str, dane: list).**

c. W funkcji, sprawdź, czy lista **dane** nie jest pusta. Jeśli jest, wyświetl komunikat i zakończ.

d. Jeśli dane istnieją, pobierz nazwy kolumn z kluczy pierwszego słownika na liście (**dane[0].keys()).**

e. Użyj bloku **with** i stwórz **csv.DictWriter,** przekazując mu nazwy pól.

f. Zapisz nagłówek **(.writeheader())** i wszystkie dane **(.writerows()).**

g. Wywołaj funkcję i sprawdź, czy plik **raport.csv** został poprawnie utworzony.

Omówienie Zadania 4 - Zapisywanie Plików CSV (csv.DictWriter)

Cel: Analiza rozwiązania i zrozumienie, jak dynamicznie i bezpiecznie generować pliki CSV.

Kluczowe mechanizmy:

- **Dynamiczne nagłówki: dane[0].keys()** pozwala na elastyczność - funkcja zadziała dla dowolnych słowników.
- **Porządek: csv.DictWriter** używa **fieldnames** do ustalenia kolejności kolumn.
- **Kompletność: writeheader() i writerows()** to dwa kluczowe kroki do stworzenia pełnego pliku CSV.

Zadanie 5 - Czytanie i Parsowanie JSON (json.load)

Problem: Jak wczytać plik z danymi o złożonej, zagnieżdżonej strukturze i przekształcić go w użyteczny obiekt w Pythonie?

Rozwiązanie: json.load() - parsuje plik tekstowy zawierający JSON i konwertuje go na natywne struktury Pythona (słowniki i listy).

Zadanie: Napisz funkcję **wczytaj_konfiguracje(sciezka),** która wczyta plik konfiguracyjny aplikacji.

a. **Przygotuj plik:** Stwórz plik **konfiguracja.json** z treścią:

```
{
  "nazwa_hosta": "localhost",
  "port": 8080,
  "debug_mode": true,
  "baza_danych": {
    "uzytkownik": "admin",
    "haslo": "super_tajne_haslo_123"
  },
  "wspierane_api": [
    "users",
    "products"
  ]
}
```

```
# Wzorzec do użycia:
import json

with open("plik.json", "r", encoding="utf-8") as f:
    dane_jako_slownik = json.load(f)
    # dane_jako_slownik to teraz normalny słownik!
    print(dane_jako_slownik['klucz'])
    print(dane_jako_slownik['zagniezdzony_obiekt']['inny_klucz'])
```

b. **Napisz funkcję: wczytaj_konfiguracje(sciezka_pliku: str) -> dict.**

c. W funkcji, użyj bloku **try...except** do obsługi:

- **FileNotFoundError:** Jeśli plik nie istnieje.
- **json.JSONDecodeError:** Jeśli plik ma niepoprawny format JSON.

d. Jeśli wczytanie się powiedzie, funkcja ma zwrócić sparsowany słownik. W razie błędu, ma zwrócić pusty słownik **{}**.

e. **Przetestuj:** Wywołaj funkcję, a następnie z wczytanych danych wyświetl nazwę użytkownika bazy danych (**dane['baza_danych']['uzytkownik']**).

Omówienie Zadania 5 - Czytanie i Parsowanie JSON (json.load)

Cel: Analiza rozwiązania i zrozumienie, jak budować solidny parser JSON odporny na błędy.

Kluczowe mechanizmy:

- **json.load(plik):** Parsuje otwarty plik i konwertuje go na słowniki/listy.
- **except FileNotFoundError:** Obsługuje przypadek, gdy plik nie istnieje.
- **except json.JSONDecodeError:** Obsługuje przypadek, gdy plik jest uszkodzony lub nie jest poprawnym JSON-em.

Zadanie 6 - Zapisywanie do JSON (json.dump)

Problem: Jak przekonwertować pythonowy słownik na czytelny dla człowieka plik JSON, poprawnie obsługując polskie znaki?

Rozwiązanie: json.dump() z odpowiednimi argumentami.

Mini-pokaz (jak to działa):

```
import json

dane_do_zapisu = {"uzytkownik": "gżegżółka", "id": 123}

with open("uzytkownik.json", "w", encoding="utf-8") as f:
    json.dump(
        dane_do_zapisu,
        f,
        indent=4,          # Tworzy "ładny" plik z wcięciami
        ensure_ascii=False  # Kluczowe dla polskich znaków!
    )
```

Zadanie: Napisz funkcję **zapisz_jako_json(dane, sciezka),** która zapisze dowolny obiekt (słownik/listę) do pliku JSON.

a. **Przygotuj dane:** Stwórz słownik **moje_dane** z kilkoma kluczami, w tym jednym z polskimi znakami (np. **"ulubiony_kolor": "żółty").**

b. **Napisz funkcję: zapisz_jako_json(dane: dict | list, sciezka_pliku: str).**

c. W funkcji, użyj bloku **try...except IOError** na wypadek problemów z zapisem na dysku.

d. W bloku **try,** otwórz plik z **with** i **encoding="utf-8".**

e. Użyj **json.dump()** do zapisu danych. **Pamiętaj** o argumentach **indent=4** i **ensure_ascii=False.**

f. Jeśli zapis się powiedzie, wyświetl komunikat o sukcesie. W razie błędu, wyświetl komunikat o błędzie.

g. **Przetestuj:** Wywołaj funkcję i sprawdź, czy plik **dane.json** został poprawnie utworzony, jest czytelny i zawiera polskie znaki.

Omówienie Zadania 6 - Zapisywanie do JSON (json.dump)

Cel: Analiza rozwiązania i zrozumienie, jak tworzyć czytelne i poprawne pliki JSON.

Kluczowe mechanizmy:

- **json.dump(obiekt, plik, ...):** Serializuje obiekt Pythona do pliku.
- **indent=4:** Tworzy "ładny" plik z wcięciami, co drastycznie poprawia czytelność dla człowieka.
- **ensure_ascii=False:** Gwarantuje, że znaki spoza ASCII (np. **ż, ą**) są zapisywane bezpośrednio, a nie jako sekwencje **\uXXXX.**

Zadanie 7 - Walidacja Danych z Pydantic

Problem: json.load() daje nam "głupi" słownik. Nie mamy żadnej gwarancji:

- Czy wszystkie klucze istnieją? (**dane['adres'] -> KeyError**)
- Czy typy danych są poprawne? (**dane['wiek']** to na pewno **int**?)

Rozwiązanie: Pydantic - biblioteka, która zamienia "głupie" słowniki na "inteligentne", bezpieczne obiekty.

```
from pydantic import BaseModel, ValidationError

class UzytkownikModel(BaseModel):
    imie: str
    wiek: int
    aktywny: bool = True # Wartość domyślna

dane_z_json = {"imie": "Jan", "wiek": "42"}

try:
    # 3. Pydantic parsuje, waliduje i konwertuje typy!
    uzytkownik = UzytkownikModel.parse_obj(dane_z_json)
    print(uzytkownik)
    # > imie='Jan' wiek=42 aktywny=True
    print(f"Wiek: {uzytkownik.wiek} (typ: {type(uzytkownik.wiek)})")
except ValidationError as e:
    print(f"Błąd walidacji: {e}")
```

Zadanie: Stwórz system walidacji konfiguracji produktu z pliku JSON.

a. **Przygotuj plik produkt.json:**

```
{
  "nazwa_produktu": "Smartfon XYZ",
  "id_produktu": "prod-12345",
  "cena": "1999.99",
  "dostepny": true,
  "tagi": ["elektronika", "nowość"],
  "specyfikacja": {
    "procesor": "SuperChip 1000",
    "ram_gb": 8
  }
}
```

b. **Zdefiniuj modele Pydantic:**

- Stwórz klasę SpecyfikacjaModel(**BaseModel**) z polami procesor: str i ram_gb: int.
- Stwórz klasę ProduktModel(**BaseModel**) z polami: nazwa_produktu: str, id_produktu: str, cena: float, dostepny: bool, tagi: list[str] oraz specyfikacja: SpecyfikacjaModel.

a. Napisz funkcję wczytaj_i_waliduj_produkt(sciezka: str) -> ProduktModel | None:

- Funkcja ma wczytać plik JSON i użyć **ProduktModel.parse_obj()** do walidacji wczytanych danych.
- W bloku **try...except** obsłużyć błędy: **FileNotFoundError, json.JSONDecodeError** oraz **pydantic.ValidationError**. W razie błędu, funkcja ma wyświetlić komunikat i zwrócić **None**.
- Jeśli walidacja się powiedzie, zwrócić stworzony obiekt.

b. **Przetestuj:** Wywołaj funkcję i jeśli zwróci obiekt, wyświetl jego atrybuty, np. **produkt.nazwa_produktu** i **produkt.specyfikacja.procesor**.

Omówienie Zadania 7 - Walidacja Danych z Pydantic

Cel: Analiza rozwiązania i zrozumienie, jak zbudować kompletny i solidny potok wczytywania i walidacji danych.

Kluczowe mechanizmy:

- **Deklaracja schematu:** Klasy Pydantic definiują "kontrakt" danych.
- **Automatyczna konwersja: Pydantic** sam zamienił **"1999.99"** na float.
- **Kompletna obsługa błędów:** Jeden blok **try** łapie błędy pliku, formatu i danych.

Zadanie 8 - Pliki w Pamięci (io.StringIO)

Problem: Czasem chcemy przetwarzać dane tak, jakby były plikiem, ale bez faktycznego zapisywania ich na dysku.

Rozwiązanie: Obiekty "pliko-podobne" (file-like objects) z modułu **io**.

- **io.StringIO**: Udaje plik **tekstowy** w pamięci.
- **io.BytesIO**: Udaje plik **binarny** w pamięci.

Zadanie: Wygeneruj raport CSV w całości w pamięci RAM.

a. Zaimportuj moduły **io** i **csv**.

b. Stwórz funkcję **generuj_raport_csv_w_pamieci(dane: list[dict]) -> io.StringIO**.

c. Wewnątrz funkcji, stwórz obiekt **plik_w_pamieci = io.StringIO()**.

d. Użyj **csv.DictWriter**, aby zapisać dane do tego obiektu, tak jakby był normalnym plikiem.

e. **Kluczowy krok:** Po zapisie, "przewiń" plik w pamięci na początek za pomocą **plik_w_pamieci.seek(0).**

f. Zwróć obiekt **plik_w_pamieci**.

g. **Przetestuj:**

- Przygotuj listę słowników z danymi.
- Wywołaj swoją funkcję, aby dostać "plik" w pamięci.
- Przekaż ten obiekt do innej funkcji (np. **przetworz_raport(plik)),** która po prostu wczyta jego zawartość za pomocą **.read()** i ją wyświetli.

Omówienie Zadania 8 - Pliki w Pamięci (io.StringIO)

Cel: Analiza rozwiązania i zrozumienie potęgi obiektów "pliko-podobnych".

Kluczowe mechanizmy:

- **Abstrakcja:** Funkcje takie jak **csv.DictWriter** czy nasza **przetworz_raport** nie wymagają *prawdziwego pliku*. Wymagają obiektu, który ma odpowiednie metody (**.write(), .read()**).
- **io.StringIO:** Udaje plik tekstowy, ale operuje na stringu w pamięci.
- **.seek(0):** Po zapisie, "kursor" jest na końcu. **seek(0)** przesuwa go na początek, umożliwiając odczyt.
- **Zastosowanie:** Unikanie tworzenia plików tymczasowych, testowanie kodu operującego na plikach, praca z API.

Zadanie 9 - Profesjonalne Logowanie (logging)

Problem: Używanie **print()** do debugowania i raportowania błędów jest złą praktyką.

Nie ma poziomów ważności, nie da się go łatwo wyłączyć

Rozwiązanie: Moduł logging - standardowy sposób na "instrumentację" aplikacji.

```
import logging

# 1. Prosta konfiguracja na początku programu
logging.basicConfig(
    level=logging.INFO, # Minimalny widoczny poziom
    format='%(asctime)s - %(levelname)s - %(message)s',
    filename='app.log', # Zapis do pliku
    filemode='w'        # Nadpisuj plik przy uruchomieniu
)

logging.info("Program wystartował.")
logging.warning("To jest ostrzeżenie.")
try:
    1 / 0
except ZeroDivisionError:
    # 2. Poprawne logowanie wyjątków z pełnym tracebackiem
    logging.error("Wystąpił błąd!", exc_info=True)
```

pliku.

a. Zaimportuj moduł **logging**.

b. Skonfiguruj **logging** za pomocą **basicConfig** tak, aby zapisywał logi do pliku **parser.log** w trybie nadpisywania (**'w'**). Ustaw poziom na **DEBUG** i zdefiniuj format.

c. Przygotuj listę **linie_do_przetworzenia** zawierającą kilka stringów, np.: *["INFO: Uruchomiono system.", "DEBUG: Sprawdzam status.", "WARNING: Niski poziom baterii.", "ERROR: Nie można połączyć z serwerem."]*

d. Napisz funkcję **przetworz_logi(linie: list)**.

e. Wewnątrz funkcji, iteruj po każdej linii:

- Użyj **if/elif/else** i metody **.startswith(),** aby sprawdzić, od jakiego słowa kluczowego zaczyna się linia.
- Jeśli linia zaczyna się od **"INFO:",** zaloguj resztę wiadomości (bez prefiksu) z poziomem **logging.INFO**.
- Analogicznie dla **"DEBUG:", "WARNING:" i "ERROR:".**
- Jeśli linia nie pasuje do żadnego wzorca, zaloguj ją z poziomem **logging.WARNING** jako "Nierozpoznana linia".

f. Wywołaj funkcję i po jej wykonaniu sprawdź zawartość pliku **parser.log**.

Omówienie Zadania 9 - Profesjonalne Logowanie (logging)

Cel: Analiza rozwiązania i zrozumienie, jak zastąpić print() uporządkowanym systemem logowania.

Podsumowanie: Kompletny Warsztat Pracy z Danymi

Co Osiągnęliśmy?

- **Nowoczesna praca z plikami:** Od obiektowych ścieżek (pathlib) po bezpieczny zapis i odczyt.
- **Serializacja:** Zapisywanie i odczytywanie całych obiektów Pythona (pickle).
- **Dane tabelaryczne:** Profesjonalna obsługa plików CSV (csv.DictReader, csv.DictWriter).
- **Dane zagnieżdżone:** Parsowanie i generowanie plików JSON (json.load, json.dump).
- **Walidacja danych:** Zamiana "głupich" słowników na "inteligentne" obiekty (Pydantic).
- **Zaawansowane techniki:** Operacje na plikach w pamięci (io.StringIO) i profesjonalne logowanie zdarzeń (logging).

Wielki Obraz:

- Przeszliśmy drogę od prostego zapisu tekstu do budowy solidnych, odpornych na błędy potoków przetwarzania danych.
- Nasz kod potrafi teraz bezpiecznie komunikować się ze światem zewnętrznym, rozumiejąc strukturę i dbając o jakość danych.