Interakcja ze Światem: Pliki i Błędy

Gdzie jesteśmy?

- Opanowaliśmy programowanie obiektowe. Potrafimy projektować spójne wewnętrznie systemy.

Nowy problem:

- Nasze programy żyją w "bańce". Po ich zamknięciu wszystkie dane znikają.
- Nasz kod jest "optymistyczny" - zakłada, że wszystko pójdzie dobrze.

Cel na dziś: Nauczyć nasze programy **trwale zapisywać dane i elegancko radzić sobie z błędami.**

Plan Działania:

a. **Zadanie 1: Podstawy Operacji na Plikach.**
   - Zapisywanie i odczytywanie plików tekstowych **(open, write, read).**
   - Zrozumienie trybów **('r', 'w', 'a').**

b. **Zadanie 2: Obsługa Wyjątków (try...except)**
   - Napiszemy kod, który nie "wykrzacza się", gdy plik nie istnieje.
   - Obsłużymy błędy konwersji typów.

c. **Zadanie 3: Menedżery Kontekstu (with)**
   - Poznamy nowoczesny i bezpieczny sposób pracy z plikami.

d. **Zadanie 4: Rzucanie Własnych Wyjątków (raise)**
   - Stworzymy własne, semantyczne błędy dla naszej logiki.

e. **Zadanie 5: Własny Menedżer Kontekstu (__enter__, __exit__)**

f. **Zadanie 6: Wzorzec "Bezpieczny Zapis" (Transakcyjność)**

g. **Zadanie 7: Praktyczny Parser Logów**

h. **Zadanie 8: Zaawansowany Menedżer Kontekstu (Tłumienie Wyjątków)**

i. **Zadanie 9: Pułapka Modyfikacji Pliku w Miejscu (seek, r+)**

j. **Podsumowanie.**

Zadanie 1 - Podstawy Operacji na Plikach

Problem: Jak trwale zapisać dane (np. wpis do dziennika) i odczytać je po ponownym uruchomieniu programu?

Narzędzia:

- **open(sciezka, tryb):** Otwiera plik.
- **plik.write(tekst):** Zapisuje tekst do pliku.
- **plik.read():** Odczytuje całą zawartość pliku.
- **plik.close():** Niezwykle ważne! Zamyka plik.

Zadanie: Stwórz prosty dziennik pokładowy.

a. **Zapis:**
   - Otwórz plik o nazwie **dziennik.txt** w trybie zapisu (**'w'**).
   - Zapisz w nim dwie linie tekstu, np. **"Pierwszy wpis."** i **"Wszystko działa."**. Pamiętaj o dodaniu znaku nowej linii **\n**.
   - Zamknij plik za pomocą metody **.close().**

b. **Weryfikacja zapisu:**
   - Sprawdź w eksploratorze plików, czy plik dziennik.txt faktycznie powstał i zawiera wpisany tekst.

c. **Odczyt:**
   - Otwórz ten sam plik dziennik.txt w trybie odczytu ('r').
   - Wczytaj całą jego zawartość do zmiennej za pomocą .read().
   - Wyświetl zawartość tej zmiennej.
   - Zamknij plik.

d. **Dopisywanie (Append):**
   - Otwórz plik dziennik.txt w trybie dopisywania ('a').
   - Dodaj nowy wpis, np. "Dodaję kolejną linię.".
   - Zamknij plik.
   - Ponownie odczytaj cały plik i wyświetl jego zawartość, aby zobaczyć wszystkie trzy linie.

Omówienie Zadania 1 - Podstawy Operacji na Plikach

Cel: Analiza rozwiązania i podkreślenie, jak łatwo zapomnieć o **plik.close().**

Kluczowe obserwacje:

- Tryb **'w'** nadpisał plik od zera.
- Tryb **'a'** dodał treść na końcu, nie kasując starej.
- **Problem:** Co, jeśli między **open()** a **close()** wystąpi błąd? Plik pozostanie otwarty!

Zadanie 2 - Obsługa Wyjątków (try...except)

Problem: Nasz poprzedni kod był "kruchy". Co się stanie, gdy:

- Spróbujemy odczytać plik, który nie istnieje? -> FileNotFoundError
- Spróbujemy przekonwertować tekst na liczbę, a tekst nie jest liczbą? -> ValueError

Rozwiązanie: Blok **try...except**, który pozwala "złapać" błąd i zareagować na niego, zamiast przerywać program.

Zadanie: Stwórz funkcję, która wczytuje wiek użytkownika z pliku i oblicza rok urodzenia.

a. Stwórz plik **wiek.txt** i wpisz do niego np. **25**.

b. Napisz funkcję **oblicz_rok_urodzenia(sciezka_pliku).**

c. Wewnątrz funkcji, umieść całą logikę w bloku **try**.

d. **Otwórz plik** i wczytaj jego zawartość.

e. **Przekonwertuj** wczytany tekst na **int**.

f. Oblicz i wyświetl przybliżony rok urodzenia (aktualny rok - wiek).

g. Dodaj blok **except FileNotFoundError**:, który wyświetli komunikat "BŁĄD: Nie znaleziono pliku!".

h. Dodaj blok **except ValueError**:, który wyświetli komunikat "BŁĄD: Zawartość pliku nie jest poprawną liczbą!".

i. **Przetestuj** działanie funkcji w trzech scenariuszach:

- Gdy wszystko jest w porządku (**wiek.txt** istnieje i zawiera liczbę).
- Gdy plik nie istnieje (podaj złą ścieżkę).
- Gdy plik zawiera tekst, a nie liczbę (zmień zawartość **wiek.txt** na "abc").

Omówienie Zadania 2 - Obsługa Wyjątków

Cel: Analiza rozwiązania i utrwalenie, jak try...except buduje odporność programu na błędy.

Kluczowe obserwacje:

- Program nie przerywa działania, tylko informuje o błędzie.
- Osobne bloki **except** pozwalają na różną reakcję na różne błędy.
- Blok **finally** jest potrzebny, aby **zagwarantować** zamknięcie pliku.

Zadanie 3 - Menedżery Kontekstu (with)

Problem: Ręczne zamykanie plików w bloku finally jest rozwlekłe i łatwo o tym zapomnieć.

```
# Poprawne, ale niewygodne
plik = open("dane.txt", "r")
try:
    # ... operacje na pliku ...
finally:
    plik.close()
```

Rozwiązanie: Instrukcja with - "pythoniczny" standard pracy z zasobami, który gwarantuje ich zamknięcie.

Składnia: with open(...) as nazwa_zmiennej:

Zadanie: Przepisz kod z Zadania 1 (zapis, odczyt, dopisywanie) używając instrukcji **with**.

a. **Zapis z with:**
   - Użyj **with open("dziennik_with.txt", "w") as plik:** do zapisu kilku linii.
   - Zauważ, że nie ma potrzeby pisać **plik.close().**

b. **Odczyt z with:**
   - Użyj **with open("dziennik_with.txt", "r") as plik:** do odczytania całej zawartości.

c. **Dopisywanie z with:**
   - Użyj **with open("dziennik_with.txt", "a") as plik:** do dopisania nowej linii.

d. **Weryfikacja:** Ponownie odczytaj plik i wyświetl jego ostateczną zawartość.

Omówienie Zadania 3 - Menedżery Kontekstu (with)

Cel: Analiza rozwiązania i utrwalenie, dlaczego with jest standardem w pracy z zasobami.

Kluczowe korzyści:

- **Bezpieczeństwo:** Gwarancja zamknięcia pliku, nawet w razie błędu.
- **Czytelność:** Kod jest krótszy i bardziej przejrzysty.
- **Wygoda:** Nie trzeba pamiętać o **plik.close().**

Zadanie 4 - Rzucanie Własnych Wyjątków (raise)

Problem: Standardowe wyjątki (ValueError) są ogólne. Czasem chcemy zasygnalizować błąd specyficzny dla naszej logiki (np. "Próba dodania ujemnej liczby produktów do koszyka").

Rozwiązanie:

a. **Tworzenie własnych klas wyjątków**: Najlepsza praktyka. Tworzymy własne klasy, które dziedziczą po Exception. To sprawia, że kod jest bardziej semantyczny.

b. **Rzucanie wyjątków (raise):** Świadomie "rzucamy" wyjątek, aby zasygnalizować błąd.

```
class NazwaTwojegoBledu(Exception):
    pass

raise NazwaTwojegoBledu("Komunikat błędu")
```

Zadanie: Zaimplementuj walidację przy dodawaniu produktu do koszyka.

a. **Stwórz wyjątek:** Stwórz nową klasę **NiepoprawnaIloscProduktuError**, która dziedziczy po **ValueError**.

b. **Stwórz funkcję:** Napisz funkcję **dodaj_do_koszyka(produkt, ilosc).**

c. Wewnątrz funkcji, dodaj walidację: jeśli **ilosc <= 0**, **podnieś (raise)** swój nowy wyjątek: **raise NiepoprawnaIloscProduktuError("Ilość produktów musi być dodatnia!").**

d. Jeśli ilość jest poprawna, niech funkcja po prostu wyświetli komunikat o dodaniu produktu.

e. **Przetestuj:**

- Wywołaj funkcję z poprawną ilością.
- Wywołaj funkcję z ujemną ilością w bloku **try...except**, który będzie łapał tylko **NiepoprawnaIloscProduktuError** i wyświetlał przyjazny komunikat.

Omówienie Zadania 4 - Rzucanie Własnych Wyjątków

Cel: Analiza rozwiązania i utrwalenie, jak tworzyć i obsługiwać niestandardowe, semantyczne błędy.

Zadanie 5: Własny Menedżer Kontekstu

Problem: Jak sprawić, by nasze własne obiekty działały z instrukcją with? Np. do automatycznego mierzenia czasu lub zarządzania połączeniem z bazą danych.

Rozwiązanie: Protokół Menedżera Kontekstu. Każdy obiekt, który ma metody **__enter__** i **__exit__,** może być użyty w **with.**

- **__enter__(self):** Wywoływana na **początku** bloku **with.**
- **__exit__(self, typ_błędu, wart_błędu, traceback):** Wywoływana **zawsze** na końcu bloku with **(do "sprzątania").**

Zadanie: Stwórz klasę **MiernikCzasu,** która będzie menedżerem kontekstu do mierzenia czasu wykonania kodu.

a. Zaimportuj moduł **time.**

b. Stwórz klasę **MiernikCzasu.**

c. Zaimplementuj metodę **__enter__(self).**

- Niech wyświetli komunikat "Rozpoczynam pomiar.".
- Niech zapisze aktualny czas **(time.perf_counter())** w atrybucie **self.start.**

d. Zaimplementuj metodę **__exit__(self, exc_type, exc_val, exc_tb).**

- Niech zapisze aktualny czas **(time.perf_counter())** w atrybucie **self.koniec.**
- Obliczy i wyświetli czas trwania operacji.

e. Przetestuj:

- Użyj swojego menedżera do zmierzenia czasu wykonania długiej pętli lub innej operacji.

```
with MiernikCzasu():
    # Kod, którego czas wykonania chcemy zmierzyć
    suma = sum(n for n in range(10_000_000))
```

Omówienie Zadania 5 - Własny Menedżer Kontekstu

Cel: Analiza rozwiązania i zrozumienie, jak protokół **__enter__ / __exit__** pozwala tworzyć własne obiekty dla instrukcji with.

Zadanie 6 - Wzorzec "Bezpieczny Zapis"

Problem: Co, jeśli program "wykrzaczy się" w połowie zapisywania ważnego pliku konfiguracja.json? Plik zostanie uszkodzony.

Rozwiązanie: Transakcyjność. Nigdy nie modyfikuj ważnego pliku bezpośrednio. Zapisuj do pliku tymczasowego, a na końcu atomowo zamień pliki.

Zadanie: Stwórz menedżer kontekstu BezpiecznyZapis, który implementuje ten wzorzec.

a. Zaimportuj moduł **os.**

b. Stwórz klasę **BezpiecznyZapis**.

c. W __init__(self, sciezka):

- Zapisz ścieżkę docelową i utwórz ścieżkę tymczasową (np. z końcówką **.tmp**).

d. W **__enter__(self):**

- Otwórz plik tymczasowy w trybie zapisu i zwróć jego uchwyt.

e. W **__exit__(self, typ_bledu, wart_bledu, traceback):**

- Zamknij plik tymczasowy.
- Sprawdź, czy **typ_bledu is None**.
- Jeśli **nie było błędu**: użyj **os.replace(),** aby zamienić plik tymczasowy na docelowy.
- Jeśli **wystąpił błąd**: użyj **os.remove(),** aby usunąć plik tymczasowy.

f. Przetestuj:

- Stwórz plik **konfiguracja.txt** z jakąś starą treścią.
- Użyj **with BezpiecznyZapis("konfiguracja.txt") as f**: i zapisz nową treść. Sprawdź, czy plik został poprawnie nadpisany.
- Przetestuj scenariusz z błędem: wewnątrz bloku **with** rzuć wyjątek i sprawdź, czy oryginalny plik **konfiguracja.txt** pozostał nietknięty.

Omówienie Zadania 6 - Wzorzec "Bezpieczny Zapis"

Cel: Analiza rozwiązania i zrozumienie, jak menedżer kontekstu

może implementować logikę transakcyjną.

Zadanie 7 - Praktyczny Parser Logów

Problem: Mamy plik z logami serwera. Każda linia ma format **POZIOM:Wiadomość.** Chcemy zliczyć, ile było błędów **(ERROR).**

Przykład pliku log.txt:

```
INFO:Uruchomiono serwer.
DEBUG:Nawiązano połączenie z bazą danych.
WARNING:Niskie miejsce na dysku.
ERROR:Nie udało się przetworzyć żądania #123.
To jest linia bez formatu.
INFO:Zakończono zadanie.
ERROR:Błąd autoryzacji użytkownika 'admin'.
```

Zadanie: Napisz funkcję **zlicz_bledy(sciezka_pliku),** która:

a. Bezpiecznie otworzy plik za pomocą **with**.

b. Będzie iterować po pliku linia po linii (bez wczytywania całości do pamięci).

c. Dla każdej linii, użyje bloku **try...except**, aby poradzić sobie z niepoprawnym formatem (np. brakiem **:**).

d. Jeśli linia ma poprawny format, sprawdzi, czy poziom to **ERROR**.

e. Zwróci całkowitą liczbę znalezionych błędów.

f. Funkcja powinna też obsłużyć **FileNotFoundError**, jeśli plik nie istnieje, i zwrócić w takim przypadku **0**.

Wskazówka: Użyj **linia.strip().split(':', 1)** do podziału linii na maksymalnie dwie części.

Omówienie Zadania 7 - Praktyczny Parser Logów

Cel: Analiza rozwiązania i zrozumienie, jak połączyć wszystkie poznane techniki w jeden, solidny program.

Kluczowe elementy solidnego kodu:

- **Odporność na brak pliku:** Zewnętrzny blok **try...except FileNotFoundError**.
- **Wydajność pamięciowa:** Iteracja **for linia in plik** zamiast **plik.read().**
- **Odporność na błędy formatu:** Wewnętrzny blok **try...except ValueError** dla każdej linii.
- **Bezpieczeństwo zasobów:** Użycie **with** gwarantuje zamknięcie pliku.

Zadanie 8 - Zaawansowany Menedżer Kontekstu (Tłumienie Wyjątków)

Problem: Czasem chcemy, aby menedżer kontekstu "połknął" pewne błędy, nie pozwalając im "wyjść" na zewnątrz bloku with.

Rozwiązanie: Metoda **__exit__** może zwrócić **True**, aby zasygnalizować, że wyjątek został obsłużony i nie powinien być dalej propagowany.

Anatomia __exit__:

```
def __exit__(self, typ_bledu, wart_bledu, traceback):
    # ... logika sprzątająca ...
    if typ_bledu is MojOczekiwanyBlad:
        print("Obsłużyłem błąd, nie propaguję go dalej.")
        return True # Tłumimy wyjątek
    return False # Nie tłumimy, wyjątek "wyjdzie" z 'with'
```

Zadanie: Stwórz menedżer kontekstu **IgnorujBledy**, który będzie ignorował określone typy błędów.

a. Stwórz klasę **IgnorujBledy**.

b. W **__init__** przyjmij krotkę typów błędów do zignorowania, np. **self.bledy_do_ignorowania = (ValueError, TypeError).**

c. **__enter__** może być puste **(pass).**

d. W **__exit__** zaimplementuj logikę:

- Sprawdź, czy **typ_bledu** (pierwszy argument) jest podklasą któregoś z błędów na liście do ignorowania (**issubclass**).
- Jeśli tak, wyświetl komunikat o zignorowaniu błędu i zwróć **True**.
- W przeciwnym razie, zwróć **False** (lub nic).

e. **Przetestuj:**

- Użyj **with IgnorujBledy((ValueError, TypeError)):** i wewnątrz bloku spróbuj wykonać **int("abc").** Sprawdź, czy program się nie "wykrzacza".
- Użyj tego samego menedżera i wewnątrz bloku spróbuj wykonać **1 / 0**. Sprawdź, czy błąd **ZeroDivisionError** jest normalnie rzucany.

Omówienie Zadania 8 - Zaawansowany Menedżer Kontekstu

Cel: Analiza rozwiązania i zrozumienie, jak metoda **__exit__** może "tłumić" wyjątki.

Kluczowe mechanizmy:

- Metoda **__exit__** otrzymuje informacje o wyjątku (**typ_bledu, wart_bledu, traceback**).
- Jeśli **__exit__** zwróci **True**, wyjątek jest stłumiony i nie propaguje się dalej.
- Jeśli **__exit__** zwróci cokolwiek innego (np. **False** lub **None**), wyjątek jest **normalnie rzucany** po wyjściu z bloku **with.**

Zadanie 9 - Pułapka Modyfikacji Pliku w Miejscu (seek, r+)

Problem: Jak zmodyfikować fragment pliku bez przepisywania go w całości?

Narzędzia:

- Tryb **r+**: Otwiera plik do odczytu i zapisu. Kursor na początku.
- **plik.tell():** Zwraca aktualną pozycję "kursora" w pliku.
- **plik.seek(pozycja):** Przesuwa "kursor" na daną pozycję.

Zadanie (Eksperyment): Mamy plik **szablon.txt** z treścią **Witaj, [IMIE]!**. Napisz funkcję **podmien_imie(sciezka, nowe_imie),** która spróbuje podmienić **[IMIE]** na **nowe_imie**.

a. Stwórz plik **szablon.txt** z treścią **Witaj, [IMIE]!.**

b. W funkcji, otwórz plik w trybie **r+** (użyj **with**).

c. Wczytaj całą zawartość pliku do zmiennej **(.read()).**

d. Użyj **zmienna.replace('[IMIE]', nowe_imie),** aby stworzyć nową treść.

e. "Przewiń" plik na początek za pomocą **plik.seek(0).**

f. Zapisz nową treść do pliku za pomocą **plik.write().**

g. Przetestuj funkcję dwa razy:

- Z **nowe_imie = "Anna"** (krótsze niż **[IMIE]).**
- Z **nowe_imie = "Krzysztof**" (dłuższe niż **[IMIE]).**

h. Obserwacja: Sprawdź zawartość pliku po każdym teście. Co się stało? Dlaczego?

Omówienie Zadania 9 - Pułapka Modyfikacji Pliku w Miejscu

Cel: Analiza wyników eksperymentu i zrozumienie, dlaczego modyfikacja plików "w miejscu" jest ryzykowna.

Obserwacje:

- Wynik dla 'Anna': **'Witaj, Anna!IE]!'**
- Wynik dla 'Krzysztof': **'Witaj, Krzysztof!'**

Wyjaśnienie:

- **plik.write() nadpisuje** bajty od aktualnej pozycji kursora.
- Nie **skraca** automatycznie pliku, jeśli nowa treść jest krótsza.
- W przypadku "Anny", słowo **Anna** nadpisało [IMI, ale reszta oryginalnej treści (E]!) pozostała w pliku.

Poprawne rozwiązanie: Po zapisie nowej treści, należałoby jawnie "obciąć" plik do nowej długości za pomocą **plik.truncate().**

Wniosek: Modyfikacja w miejscu jest skomplikowana. Bezpieczniejszym i prostszym wzorcem jest **wczytanie, modyfikacja w pamięci i zapisanie całości od nowa** (np. do pliku tymczasowego, jak we wzorcu "Bezpieczny Zapis").

Podsumowanie – Fundamenty Solidnego Kodu

Co Osiągnęliśmy?

- Nasze programy potrafią trwale przechowywać dane w plikach.
- Opanowaliśmy **kompletny mechanizm obsługi błędów (try...except, raise,** własne wyjątki**).**
- Umiemy **bezpiecznie zarządzać zasobami** dzięki menedżerom kontekstu **(with, __enter__, __exit__).**
- Zrozumieliśmy **zaawansowane wzorce i pułapki** (transakcyjny zapis, tłumienie wyjątków, modyfikacja w miejscu).

Kluczowe Nawyki Profesjonalisty:

- **Zasoby w bloku with:** Zawsze używaj **with** do pracy z plikami.
- **Łap konkretne błędy:** Bądź precyzyjny w **except**, unikaj **except Exception.**
- **Sygnalizuj własne błędy:** Używaj **raise** z własnymi, semantycznymi klasami wyjątków.
- **Myśl o przypadkach brzegowych:** Co jeśli plik nie istnieje? Co jeśli jest za duży? Co jeśli ma zły format?

Co dalej? (Laboratorium 5, cz. 2) Opanowaliśmy interakcję z plikami jako strumieniami bajtów/tekstu. Na następnych zajęciach wejdziemy na wyższy poziom:

- Poznamy **nowoczesne, obiektowe podejście do pracy ze ścieżkami (**moduł **pathlib).**
- Nauczymy się **serializować obiekty (pickle).**
- Zgłębimy pracę z danymi ustrukturyzowanymi **(CSV, JSON).**