Współbieżność i Uzupełnienie Warsztatu

Gdzie jesteśmy?

Opanowaliśmy zaawansowaną pracę z plikami i danymi (CSV, JSON, Pydantic).

Nowe wyzwania:

- **Luki w wiedzy: re** (wyrażenia regularne) i **datetime** (data i czas).
- **Tyrania sekwencyjności:** Nasze programy robią jedną rzecz naraz, przez co są powolne i niereponsywne, zwłaszcza przy operacjach sieciowych (I/O-bound) lub ciężkich obliczeniach (CPU-bound).

Cel na dziś:

- **Uzupełnić warsztat** o wyrażenia regularne i pracę z czasem.
- Nauczyć programy robić wiele rzeczy **"jednocześnie" (współbieżnie).**

Plan Działania:

Część 1: Rozgrzewka - Uzupełnienie Warsztatu

- Zadanie 1: Wyrażenia Regularne w Praktyce (**re**).
- Zadanie 2: Praca z Czasem (**datetime**).

Część 2: Współbieżność I/O-bound

- Zadanie 3: Wielowątkowość (**threading**).
- Zadanie 4: Pułapka - Stan Wyścigu (**Race Condition**).
- Zadanie 5: Synchronizacja z **threading.Lock**.

Część 3: Równoległość CPU-bound

- Zadanie 6: Wieloprocesowość (**multiprocessing**).

Część 4: Nowoczesna Współbieżność

- Zadanie 7: Programowanie Asynchroniczne (**asyncio**).

Część 5: Wzorce i Abstrakcje

- Zadanie 8: Bezpieczna komunikacja (**queue.Queue**).
- Zadanie 9: Nowoczesne API (**concurrent.futures**).

Część 6: Podsumowanie

- Wybór właściwego narzędzia do współbieżności.

Zadanie 1: Wyrażenia Regularne w Praktyce (re)

Problem: Jak z pliku tekstowego (np. logu serwera) wyciągnąć wszystkie dane o określonym formacie, np. adresy IP? Proste metody .find() czy .split() są niewystarczające.

Rozwiązanie: Moduł re i jego funkcja re.findall(wzorzec, tekst).

Zadanie: Napisz funkcję, która z pliku z logami wyciągnie wszystkie adresy IP oraz kody statusu HTTP.

a. **Przygotuj plik:** Stwórz plik **log.txt** z poniższą treścią:

```
127.0.0.1 - - [28/Oct/2023:10:55:36] "GET /index.html" 200
89.161.25.13 - - [28/Oct/2023:10:56:01] "POST /login" 404
212.77.100.101 - - [28/Oct/2023:10:57:15] "GET /admin" 500
127.0.0.1 - - [28/Oct/2023:10:58:00] "GET /dashboard" 200
```

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

c. W funkcji, wczytaj całą zawartość pliku do stringa.

d. **Wzorzec dla IP:** Zdefiniuj wzorzec **r"\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"**.
Użyj **re.findall()**, aby znaleźć wszystkie adresy IP.

e. **Wzorzec dla kodu HTTP:** Zdefiniuj wzorzec **r'" (\d{3})'**. Użyj **re.findall()**, aby znaleźć wszystkie 3-cyfrowe kody statusu.

f. Funkcja powinna zwrócić słownik w formacie: **{'adresy_ip': [...], 'kody_statusu': [...]}.**

g. Wywołaj funkcję i wyświetl wynik.

Omówienie Zadania 1 - Wyrażenia Regularne w Praktyce (re)

Cel: Analiza rozwiązania i zrozumienie, jak **re.findall** upraszcza parsowanie tekstu.

Kluczowe korzyści:

- **Zwięzłość:** Dwie linie kodu (**re.findall**) zastępują skomplikowaną logikę pętli i warunków.
- **Deklaratywność:** Opisujemy, co chcemy znaleźć, a nie jak to znaleźć.
- **Elastyczność:** Łatwo dostosować wzorzec do innego formatu logów.

Zadanie 2 - Praca z Czasem (datetime)

Problem: Wyciągnęliśmy z logów adresy IP, ale co z datami? Są to tylko stringi. Jak zamienić **"28/Oct/2023:10:55:36"** na obiekt, na którym można wykonywać obliczenia (np. liczyć różnicę czasu)?

Rozwiązanie: Moduł **datetime** i jego potężna metoda **strptime** (string parse time).

- **datetime.strptime(string_z_datą, format):** Parsuje string na obiekt **datetime** według podanego formatu.

Zadanie: Rozwiń poprzednie zadanie. Napisz funkcję, która obliczy, ile czasu minęło między pierwszym a ostatnim logiem w pliku.

a. Użyj tego samego pliku **log.txt**.

b. **Napisz funkcję:**
analizuj_czas_logow(sciezka_pliku: str) -> timedelta | None.

c. **Wzorzec dla daty:** Zdefiniuj wzorzec **re** do wyciągnięcia dat, np. **r'\[(\d{2}/\w{3}/\d{4}:\d{2}:\d{2}:\d{2})\]'**. Użyj **re.findall()**.

d. **Format daty:** Zdefiniuj string formatujący dla strptime: **'%d/%b/%Y:%H:%M:%S'**.

e. Użyj list comprehension lub pętli, aby przekonwertować listę stringów z datami na listę obiektów **datetime**.

f. Jeśli lista dat jest pusta, zwróć **None**.

g. Znajdź najwcześniejszy i najpóźniejszy czas w liście (użyj **min()** i **max()**).

h. Oblicz różnicę (**timedelta = max_czas - min_czas**).

i. Funkcja powinna zwrócić ten obiekt **timedelta**.

j. Wywołaj funkcję i wyświetl wynik.

Omówienie Zadania 2 - Praca z Czasem (datetime)

Cel: Analiza rozwiązania i zrozumienie, jak połączyć **re** i **datetime** do analizy danych czasowych.

Kluczowe korzyści:

- **Konwersja:** Zamieniliśmy bezużyteczne stringi na potężne obiekty **datetime**.
- **Obliczenia:** Na obiektach **datetime** można wykonywać operacje matematyczne (**-, min, max**).
- **Synergia:** Połączenie **re** (do ekstrakcji) i **datetime** (do interpretacji) to potężny wzorzec.

Zadanie 3 - Wielowątkowość (threading)

Problem: Nasz program musi pobrać aktualne kursy walut z API NBP. Sekwencyjnie, wygląda to tak:

- **Zapytaj o USD -> CZEKAJ na sieć -> Zapytaj o EUR -> CZEKAJ -> ...**
- Całkowity czas to suma wszystkich czasów oczekiwania. Procesor przez 99% czasu się nudzi. To jest zadanie **I/O-bound**.

Rozwiązanie: Uruchomienie każdej operacji pobierania w osobnym **wątku (threading.Thread)**. Główny program zleca pracę i nie czeka, co pozwala na jednoczesne "oczekiwanie" na wiele odpowiedzi.

Zadanie: Porównaj czas pobierania kursów 5 walut w wersji sekwencyjnej i wielowątkowej.

a. Zaimportuj **requests**, **threading** i **time**.

b. Stwórz funkcję: pobierz_kurs(waluta: str). Użyj URL API NBP: *http://api.nbp.pl/api/exchangerates/rates/A/{waluta}/?format=json*
   - Wykonaj **requests.get(url)**, sparsuj JSON i wypisz kurs (pole mid).
   - Dodaj printy oznaczające początek i koniec pobierania.

c. Przygotuj dane – Lista walut: **['EUR', 'USD', 'CHF', 'GBP', 'JPY']**.

d. Część 1 - Wersja Sekwencyjna:
   - Zmierz czas (**time.perf_counter**). Użyj pętli for, aby wywołać pobierz_kurs dla każdej waluty. Zmierz i wyświetl całkowity czas.

e. Część 2 - Wersja Wielowątkowa:
   - Zmierz czas.
   - Stwórz pustą listę **watki = []**.
   - W pętli **for**, dla każdej waluty:
     - Stwórz obiekt **watek = threading.Thread(**target=pobierz_kurs, args=(waluta,)**)**.
     - Dodaj go do listy **watki**.
     - **Uruchom wątek** metodą **.start().**
   - Po pętli tworzącej, napisz drugą pętlę, która na każdym wątku z listy wywoła metodę **.join()**.
   - Zmierz i wyświetl całkowity czas wykonania. Porównaj wyniki.

Omówienie Zadania 3 - Wielowątkowość (threading)

Cel: Analiza rozwiązania i zobaczenie namacalnej różnicy w czasie wykonania.

Kluczowe korzyści:

- **Wydajność I/O:** Czas wykonania jest bliski czasowi najdłuższej operacji, a nie sumie wszystkich.
- **Responsywność:** Główny program nie jest blokowany na czas oczekiwania.

Zadanie 4 - Pułapka - Stan Wyścigu (System Rezerwacji)

Scenariusz: Mamy 5 biletów na koncert i 10 klientów (wątków) chcących je kupić w tej samej chwili.

Logika kupowania:

a. Sprawdź, czy są bilety (**bilety > 0**).

b. Jeśli tak: Czekaj na płatność (symulacja opóźnienia).

c. Zmniejsz liczbę biletów (**bilety -= 1**) i potwierdź zakup.

Problem: Wątki wchodzą w **Stan Wyścigu**.

- Wątek A sprawdza: "Są bilety?". Tak (jest 5).
- Wątek B sprawdza: "Są bilety?". Tak (wciąż jest 5, bo A jeszcze nie kupił).
- Oba wątki sprzedają bilet.

Zadanie: Symulacja "Oversellingu".

a. Zmienna globalna: **dostepne_bilety = 5**.

b. Funkcja kup_bilet(klient_id):
   - Sprawdź **if dostepne_bilety > 0:**.
   - Symuluj opóźnienie sieci/bazy: **time.sleep(0.1)**.
   - Zmniejsz licznik: **dostepne_bilety -= 1**.
   - Wypisz: **Klient {id} kupił bilet. Zostało: {bilety}.**
   - **else**: Wypisz **Klient {id} odszedł z kwitkiem..**

c. Stwórz listę 10 wątków (threading.Thread), każdy reprezentuje klienta.

d. Uruchom wszystkie wątki (start) i poczekaj na nie (join).

e. Sprawdź końcową liczbę biletów.

Omówienie Zadania 4: Katastrofa w kasie biletowej

Cel: Zrozumienie, jak brak atomowości operacji prowadzi do błędów biznesowych (sprzedaż towaru, którego nie ma).

Kluczowe wnioski:

- **Check-Then-Act:** Klasyczny błąd. Sprawdzasz warunek (**if > 0**), ale zanim podejmiesz akcję (**-= 1**), warunek przestaje być prawdziwy.
- **Konsekwencje:** Sprzedaliśmy 10 biletów, mając tylko 5.

Zadanie 5 - Synchronizacja z threading.Lock

Problem: Wątki wchodziły sobie w drogę między sprawdzeniem dostępności a zakupem.

Rozwiązanie: Blokada (**threading.Lock**). Działa jak pilot do telewizora. Tylko ten, kto go trzyma, może zmieniać kanały (modyfikować dane). Reszta musi czekać na swoją kolej.

Jak używać? Najlepiej i najbezpieczniej z instrukcją **with**, która automatycznie zarządza "chwytaniem" i "zwalnianiem" blokady.

```
blokada = threading.Lock()

with blokada:
    # Ten kod jest "bezpieczny wątkowo" (thread-safe).
    # Tylko jeden wątek może go wykonywać w danym momencie.
    ...
```

Zadanie: Napraw problem stanu wyścigu z poprzedniego zadania.

a. Skopiuj kod z zadania 4.

b. Stwórz blokadę: Na poziomie globalnym stwórz instancję blokady: blokada = threading.Lock().

c. Zabezpiecz sekcję krytyczną: W funkcji **inkrementuj**, umieśćcały blok **if…else** (sprawdzanie i zakup) wewnątrz **with blokada:**

d. Uruchom skrypt. Powinno si ę sprzedać tylko 5 biletów, a 5 klientów powinno odejść z kwitkiem. Stan końcowy: 0

Omówienie Zadania 5 - Synchronizacja z threading.Lock

Cel: Analiza rozwiązania i zrozumienie, jak Lock zapewnia bezpieczeństwo wątkowe (thread safety).

Kluczowe wnioski:

- **with blokada:** tworzy **sekcję krytyczną**.
- Tylko jeden wątek naraz może wykonywać kod w sekcji krytycznej.
- To zapobiega stanowi wyścigu i gwarantuje poprawny, przewidywalny wynik.
- Osiągnęliśmy **bezpieczeństwo wątkowe (thread safety).**

Zadanie 6 - Wieloprocesowość (multiprocessing)

Problem: threading nie przyspiesza zadań CPU-bound (ciężkich obliczeń) z powodu **Global Interpreter Lock (GIL)**. Wątki działają współbieżnie, ale nie równolegle.

Rozwiązanie: Moduł **multiprocessing**. Tworzy osobne **procesy**, każdy z własnym interpreterem Pythona i własnym GIL. Mogą one działać **prawdziwie równolegle** na wielu rdzeniach.

Narzędzie: multiprocessing.Pool - zarządza grupą procesów-robotników. Metoda **pool.map()** automatycznie rozdziela pracę i zbiera wyniki.

Zadanie: Porównaj czas wykonania ciężkich obliczeń w wersji sekwencyjnej i wieloprocesowej.

- **Stwórz funkcję-robotnika:** Napisz funkcję **ciezka_praca(n: int) -> int**, która wykonuje dużo obliczeń (np. sumuje liczby od 0 do **n**) i zwraca wynik.
- **Przygotuj dane:** Stwórz listę dużych liczb do przetworzenia, np. **[10_000_000 + i for i in range(10)]**.
- **Ważne:** Cały poniższy kod umieść w bloku **if __name__ == "__main__":**.
- **Część 1 - Wersja Sekwencyjna:**
    a. Zmierz czas (**time.perf_counter**). Użyj pętli **for** lub list comprehension, aby wywołać **ciezka_praca** dla każdej liczby. Zmierz i wyświetl całkowity czas.
- **Część 2 - Wersja Wieloprocesowa:**
    a. Zmierz czas.
    b. Użyj bloku **with multiprocessing.Pool() as pula:**.
    c. Wewnątrz bloku wywołaj wyniki = pula.map(ciezka_praca, dane**).**
    d. Zmierz i wyświetl całkowity czas wykonania. Porównaj wyniki.

Omówienie Zadania 6 - Wieloprocesowość (multiprocessing)

Cel: Analiza rozwiązania i zobaczenie realnego przyspieszenia w zadaniach CPU-bound.

Kluczowe wnioski:

- **multiprocessing omija GIL,** tworząc osobne procesy.
- **Pool.map** to najprostszy sposób na zrównoleglenie zadań typu "przetwórz tę listę".
- Blok **if __name__ == "__main__":** jest **obowiązkowy.**
- Obserwujemy **realne, znaczące przyspieszenie** na maszynach wielordzeniowych.

Zadanie 7 - Programowanie Asynchroniczne (asyncio)

Problem: threading jest świetny, ale dla **tysięcy** jednoczesnych połączeń (np. w serwerze czatu), tworzenie tysięcy wątków jest nieefektywne (duży narzut pamięci i przełączania kontekstu). Jak obsłużyć masową współbieżność I/O w jednym wątku?

Rozwiązanie: Programowanie asynchroniczne (asyncio). Współbieżność oparta na **pętli zdarzeń (event loop)** i jawnej współpracy zadań.

- **async def:** Definiuje **korutynę** - funkcję, którą można "spauzować".
- **await:** "Pauzuje" korutynę i oddaje kontrolę pętli zdarzeń, mówiąc: "Ja tu czekam na I/O, w tym czasie rób coś innego".

Uwaga: Biblioteka **requests** jest blokująca! W **asyncio** używamy biblioteki **aiohttp**. (Zainstaluj: pip install aiohttp).

Zadanie: Pobierz kursy walut (jak w Zadaniu 3), ale asynchronicznie.

a. Zaimportuj **asyncio**, **aiohttp** i **time.**

b. Korutyna pobierająca: async def pobierz_kurs(session, waluta).

- Użyj **async with session.get(url) as resp:** aby wykonać zapytanie.
- Pobierz JSON: **dane = await resp.json().**
- Zwróć sformatowany string z kursem.

c. Główna korutyna: async def main().

- Stwórz sesję HTTP: **async with aiohttp.ClientSession() as session:.**
- Stwórz listę zadań (obiektów korutyn) dla każdej waluty z listy **['EUR', 'USD', 'CHF', 'GBP', 'JPY'].**
- Uruchom je równolegle: wyniki = await asyncio.gather(*zadania).

d. Uruchom: **asyncio.run(main())**. Zmierz czas.

Omówienie Zadania 7 - Programowanie Asynchroniczne (asyncio)

Cel: Analiza rozwiązania i zrozumienie, jak **asyncio** pozwala na masową współbieżność I/O w jednym wątku.

Kluczowe korzyści:

- **Wydajność I/O na dużą skalę:** Czas wykonania jest bliski czasowi najdłuższego zadania, a nie sumie wszystkich.
- **Minimalny narzut:** Brak kosztownego przełączania kontekstu wątków przez system operacyjny.
- **Skalowalność:** Idealne do obsługi tysięcy jednoczesnych operacji I/O w jednym wątku.

Zadanie 8 - Bezpieczna komunikacja (queue.Queue)

Problem: Blokady (**Lock**) służą do ochrony danych, ale jak bezpiecznie **przekazywać** dane między wątkami? Ręczne używanie list i blokad jest trudne i podatne na błędy.

Rozwiązanie: Kolejka (queue.Queue). To struktura danych zaprojektowana specjalnie dla wielowątkowości. Jest "thread-safe" - nie wymaga ręcznego blokowania.

Wzorzec Producent-Konsument:

- **Producent:** Tworzy zadania i wkłada je do kolejki (**put**).
- **Konsument:** Pobiera zadania z kolejki (**get**) i je przetwarza.

Zadanie: Zaimplementuj prosty system przetwarzania zgłoszeń.

a. Zaimportuj moduł **queue** i **threading.**

b. Stwórz kolejkę: **kolejka = queue.Queue().**

c. Funkcja Producenta: W pętli (np. 5 razy) wygeneruj "zgłoszenie" (np. string "Zgłoszenie nr X"), włóż je do kolejki (**kolejka.put()**) i wypisz komunikat. Dodaj małe opóźnienie.

d. Funkcja Konsumenta: W nieskończonej pętli **while True:**

- Pobierz zadanie: **zadanie = kolejka.get().**
- Przetwórz je (wypisz "Przetwarzam...").
- Zasygnalizuj wykonanie: **kolejka.task_done().**

e. Uruchomienie:

- Uruchom wątek konsumenta jako **demona (daemon=True)**, aby zakończył się wraz z głównym programem.
- Uruchom wątek producenta.
- Poczekaj na opróżnienie kolejki: **kolejka.join().**

Omówienie Zadania 8 - Bezpieczna komunikacja (queue.Queue)

Cel: Analiza rozwiązania i zrozumienie, jak **queue.Queue** upraszcza bezpieczną wymianę danych między wątkami (wzorzec Producent-Konsument).

Kluczowe wnioski:

- **Brak Locków:** Nie użyliśmy ani jednego **Lock** jawnie. Kolejka robi to za nas.
- **Separacja:** Producent nie musi znać Konsumenta. Łączy ich tylko kolejka.
- **Skalowalność:** Łatwo dodać więcej wątków konsumentów, by przyspieszyć pracę.

Zadanie 9 - Nowoczesne API (concurrent.futures)

Problem: Ręczne tworzenie list wątków (**threads = []**), pętle **.start()** i **.join()** są powtarzalne i "brzydkie".

Rozwiązanie: Moduł **concurrent.futures**. To wysokopoziomowa nakładka na **threading** i **multiprocessing**.

- **ThreadPoolExecutor:** Zarządza pulą wątków.
- **ProcessPoolExecutor:** Zarządza pulą procesów.
- Interfejs jest **identyczny** dla obu!

Zadanie: Przepisz zadanie z pobieraniem URL (Zadanie 3), używając nowoczesnego **ThreadPoolExecutor**.

a. Zaimportuj **concurrent.futures**.

b. Użyj tej samej funkcji **pobierz_url** co w Zadaniu 3.

c. Zamiast ręcznych pętli, użyj menedżera kontekstu: **with concurrent.futures.ThreadPoolExecutor() as executor:**

d. Wewnątrz bloku użyj metody **executor.map(funkcja, lista_argumentow)**.

e. To wszystko! **executor** sam uruchomi wątki i poczeka na ich koniec.

Omówienie Zadania 9 - Nowoczesne API (concurrent.futures)

Cel: Analiza rozwiązania i poznanie nowoczesnego standardu zarządzania pulą wątków/procesów w Pythonie.

Kluczowe korzyści:

- **Elegancja:** Mniej kodu, mniej miejsc na błędy.
- **Elastyczność:** Łatwa zmiana backendu (wątki vs procesy).
- **Standard:** Tak pisze się nowoczesny kod współbieżny w Pythonie (jeśli nie używamy **asyncio**).

Podsumowanie - Wybór Właściwego Narzędzia

Kluczowe pytanie: Co spowalnia mój program?

- **Czekanie na sieć/dysk (I/O-bound)?** -> Potrzebujesz współbieżności.
- **Ciężkie obliczenia (CPU-bound)?** -> Potrzebujesz równoległości.

Narzędzie	Kiedy używać?	Zalety	Wady
threading	• Proste zadania I/O-bound. • Integracja ze starym, blokującym kodem.	• Proste API, współdzielona pamięć.	• GIL uniemożliwia równoległość CPU. • Ryzyko Race Condition.
multiprocessing	• Zadania CPU-bound. • Gdy musisz wykorzystać wszystkie rdzenie.	• Prawdziwa równoległość (omija GIL).	• "Ciężkie", większe zużycie pamięci. • Skomplikowana komunikacja.
concurrent.futures	• Standard dla kodu synchronicznego. • Zastępuje ręczne threading/multiprocessing.	• Proste, spójne API. • Łatwe przełączanie wątki/procesy.	• Mniejsza kontrola nad detalami niż w "surowych" modułach.
asyncio	• Zadania I/O-bound na dużą skalę. (tysiące połączeń, API, serwery)	• Najbardziej wydajne i skalowalne. • Minimalny narzut.	• "Zaraźliwe" - wymaga bibliotek async. Inny model myślenia.