Zaawansowane Iteratory i Korutyny

Gdzie jesteśmy?

- Zbudowaliśmy wydajne, jednokierunkowe potoki danych.
- Używaliśmy generatorów do produkcji danych (**yield**).
- Poznaliśmy podstawowe narzędzia **itertools** (**groupby**, **tee**).
- Wprowadziliśmy pojęcie korutyn i metody **.send()**.

Nowe wyzwania: Pełna kontrola nad strumieniem

- Jak łączyć wiele iteratorów w jeden strumień bez zużywania pamięci?
- Jak "kroić" lub filtrować strumień bez materializowania go do listy?
- Jak sterować korutyną z zewnątrz (np. resetować ją) w trakcie działania?
- Jak zbudować zaawansowaną **korutynę** (maszynę stanu) do przetwarzania poleceń?

Cel na dziś:

- Opanować zaawansowane techniki pracy ze strumieniami danych.
- Zrozumieć, jak generatory mogą stać się prostymi, interaktywnymi korutynami.

Plan Działania:

a. **Rozszerzamy warsztat itertools:**
   - Zadanie 1: Łączenie i krojenie strumieni (**chain**, **islice**).
   - Zadanie 2: Warunkowe filtrowanie strumieni (**takewhile**, **dropwhile**).

b. **Generatory jako Korutyny:**
   - Zadanie 3: Zewnętrzne sterowanie (**.throw()**) i wzorzec dekoratora.
   - Zadanie 4: Zadanie podsumowujące - Procesor Poleceń.

c. **Protokół Iteratora "pod maską":**
   - Zadanie 5: Budowa własnych, reużywalnych iteratorów.
   - Zadanie 6: Pułapka iteratora jednorazowego użytku.

d. **Podsumowanie.**

Łączenie i Krojenie Strumieni (chain, islice)

Problem:

- Jak połączyć dane z kilku różnych źródeł (np. dwóch plików, listy i generatora) w jeden, "leniwy" strumień?
- Jak wziąć tylko kilka pierwszych elementów z bardzo długiego lub nieskończonego iteratora, nie wczytując go całego?

Rozwiązanie: itertools

- **itertools.chain(*iterables):** Bierze kilka iteratorów i zwraca jeden, który produkuje elementy z pierwszego, potem z drugiego, itd.
- **itertools.islice(iterable, stop)** lub **(start, stop, step):** Działa jak "krojenie" **[:]** na listach, ale dla dowolnego iteratora i jest "leniwe".

Zadanie:

a. **Przygotuj dane:** Stwórz dwa generatory: **generator_liter** produkujący litery od 'a' do 'c' i **generator_liczb** produkujący liczby od 1 do 3.

b. **Połącz strumienie:** Użyj **itertools.chain**, aby połączyć oba generatory. Przeiteruj po wyniku i wyświetl elementy.

c. **Stwórz nieskończony strumień:** Napisz generator **nieskonczone_liczby()**, który w pętli **while True** będzie **yield**-ował kolejne liczby całkowite, zaczynając od 0.

d. **Wytnij kawałek:** Użyj **itertools.islice**, aby wziąć 10 elementów (od 5 do 14) z **nieskonczone_liczby()**. Przeiteruj po wyniku i wyświetl elementy.

Omówienie Zadania 1 - Łączenie i Krojenie Strumieni

Cel: Analiza rozwiązania i zrozumienie, jak chain i islice pozwalają na "leniwe" operacje na strumieniach.

Kluczowe wnioski:

- **Leniwość chain: chain** nie zużywa iteratorów od razu. Najpierw w pełni wyczerpuje pierwszy, a dopiero potem zaczyna pracę z drugim.
- **Bezpieczeństwo islice: islice** pozwala bezpiecznie pracować na nieskończonych iteratorach, pobierając tylko tyle elementów, ile potrzebujemy.

```
import itertools

def generator_liter():
    print("(Generator liter startuje)")
    for litera in ['a', 'b', 'c']:
        yield litera
    print("(Generator liter kończy)")

def generator_liczb():
    print("(Generator liczb startuje)")
    for liczba in [1, 2, 3]:
        yield str(liczba)
    print("(Generator liczb kończy)")

# Generatory jeszcze nie wystartowały!
polaczony_strumien = itertools.chain(generator_liter(),
generator_liczb())
print("Połączony strumień:", list(polaczony_strumien))

def nieskonczone_liczby():
    liczba = 0
    while True:
        yield liczba
        liczba += 1

# Generator nie jest uruchamiany w całości.
kawalek_strumienia = itertools.islice(nieskonczone_liczby(), 5, 15)
print("\nKawałek nieskończonego strumienia:", list(kawalek_strumienia))
```

Zadanie 2 - Warunkowe Filtrowanie (takewhile, dropwhile)

Problem: Jak "leniwie" przetwarzać strumień, ale tylko do momentu, gdy jakiś warunek przestanie być spełniany? Albo jak pominąć początkowe elementy, które spełniają warunek?

Rozwiązanie: itertools

- **itertools.takewhile(predicate, iterable):** Zwraca elementy, dopóki predykat (funkcja warunkowa) jest prawdziwy. **Zatrzymuje się przy pierwszym fałszu.**
- **itertools.dropwhile(predicate, iterable):** Omija elementy, dopóki predykat jest prawdziwy. **Zwraca całą resztę strumienia od pierwszego fałszu.**

Zadanie:

a. **Przygotuj dane:** Mamy listę odczytów z czujnika: **odczyty = [12, 15, 18, 21, 25, 19, 17]**.

b. **Problem 1:** Chcemy wziąć tylko początkowe odczyty, które są poniżej progu alarmowego (np. 22). Użyj **itertools.takewhile**.

c. **Problem 2:** Chcemy zignorować początkowe, stabilne odczyty (np. poniżej 18) i przetwarzać dane dopiero od pierwszego "skoku". Użyj **itertools.dropwhile**.

Omówienie Zadania 2 - Warunkowe Filtrowanie

Cel: Analiza rozwiązania i zrozumienie różnicy między **takewhile** a **dropwhile**.

Kluczowe wnioski:

- **takewhile:** Idealne do brania "nagłówka" strumienia.
- **dropwhile:** Idealne do pomijania "nagłówka" strumienia.
- Oba narzędzia są "leniwe" i niezwykle ekspresyjne.

```
import itertools

odczyty = [12, 15, 18, 21, 25, 19, 17]
prog_alarmowy = 22
prog_stabilny = 18

# 1. Weź odczyty, dopóki są poniżej progu alarmowego
# takewhile zatrzymuje się przy pierwszym elemencie, który nie spełnia
# warunku (25), i ignoruje resztę.
bezpieczne_odczyty = itertools.takewhile(
    lambda x: x < prog_alarmowy, odczyty
)
print(f"Bezpieczne odczyty: {list(bezpieczne_odczyty)}")
# Wynik: [12, 15, 18, 21]

# 2. Pomiń początkowe stabilne odczyty
# dropwhile pomija elementy, dopóki warunek jest prawdziwy (12, 15).
# Zaczyna zwracać od pierwszego elementu, który nie spełnia warunku
(18).
niestabilne_odczyty = itertools.dropwhile(
    lambda x: x < prog_stabilny, odczyty
)
print(f"Odczyty od pierwszego skoku: {list(niestabilne_odczyty)}")
# Wynik: [18, 21, 25, 19, 17]
```

Zadanie 3 - Pełna Kontrola - Obsługa Błędów (.throw())

Problem: Jak z zewnątrz zasygnalizować korutynie wyjątkową sytuację (np. potrzebę resetu, błąd danych) bez wysyłania specjalnych wartości?

Usprawnienie: Ręczne wywoływanie **next()** ("priming") jest niewygodne. Możemy to zautomatyzować za pomocą **dekoratora**.

Rozwiązanie: Metoda .throw(TypWyjatku)

- Wznawia generator, ale zamiast wysyłać wartość, "wstrzykuje" wyjątek w miejsce, gdzie generator jest zapauzowany (**yield**).
- Korutyna może ten wyjątek złapać za pomocą standardowego bloku **try...except**.

Zadanie: Stwórz korutynę **srednia_kroczaca**, którą można resetować z zewnątrz.

a. **Stwórz dekorator @korutyna:** Funkcja, która bierze generator, tworzy go, wywołuje **next()** i zwraca gotowy obiekt.

b. **Stwórz własny wyjątek: class ResetKorutyny(Exception): pass.**

c. **Napisz korutynę:**

   - Użyj dekoratora **@korutyna**.
   - W pętli **while True**, opakuj linię **nowa_liczba = yield** w blok **try...except ResetKorutyny:**.
   - W sekcji **except**, zresetuj **sumę** i **licznik** do zera i wydrukuj komunikat o resecie.

d. Przetestuj:

   - Stwórz instancję korutyny.
   - Wyślij kilka liczb za pomocą **.send()**.
   - Wywołaj **kalkulator.throw(ResetKorutyny)**.
   - Wyślij kolejne liczby i sprawdź, czy średnia jest liczona od nowa.

Omówienie Zadania 3
Pełna Kontrola

Cel: Analiza rozwiązania i utrwalenie, jak **.throw()** pozwala na zewnętrzne sterowanie przepływem korutyny.

Kluczowe wnioski:

- Metoda .throw() "wstrzykuje" wyjątek w miejsce, gdzie korutyna jest zapauzowana (na **yield**).
- Korutyna może złapać ten wyjątek za pomocą standardowego bloku **try...except**, co pozwala na eleganckie sterowanie jej stanem.
- **Wzorzec Dekoratora:** Automatyzacja "primingu" (pierwszego next()) czyni kod używającym korutyny znacznie czystszym.
- Oddzielamy w ten sposób dane (.send()) od sygnałów sterujących (.throw()).

```
from functools import wraps

def korutyna(func):
    @wraps(func)
    def primer(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)
        return gen
    return primer

class ResetKorutyny(Exception):
    pass

@korutyna
def srednia_kroczaca_z_resetem():
    print("Korutyna gotowa do pracy.")
    suma = 0.0
    licznik = 0
    while True:
        try:
            nowa_liczba = yield
            suma += nowa_liczba
            licznik += 1
            print(f"Otrzymano: {nowa_liczba}, nowa średnia: {suma / licznik:.2f}")
        except ResetKorutyny:
            print("--- RESETOWANIE KORUTYNY ---")
            suma = 0.0
            licznik = 0

kalkulator = srednia_kroczaca_z_resetem()
kalkulator.send(10)
kalkulator.send(20) # Średnia: 15.0
kalkulator.throw(ResetKorutyny)

print("\n>>> Wysyłam nowe dane...")
kalkulator.send(5)  # Średnia powinna być teraz 5.0
kalkulator.send(15) # Średnia powinna być teraz 10.0
```

Zadanie 4 - Procesor Poleceń (Zadanie Podsumowujące)

Cel: Zintegrowanie całej wiedzy o iteratorach, generatorach i korutynach w jednym, praktycznym programie.

Problem: Stwórz korutynę, która będzie działać jak prosty procesor poleceń. Będzie ona przyjmować polecenia (jako stringi) i odpowiednio na nie reagować.

Logika Procesora:

a. Korutyna utrzymuje wewnętrzny stan: listę **dane = []**.

b. Przyjmuje polecenia za pomocą **.send()**. Polecenie to string w formacie "**AKCJA:WARTOŚĆ**", np. "**DODAJ:jabłko**".

c. Obsługiwane akcje:

- **DODAJ:wartość**: dodaje **wartość** do listy **dane**.
- **USUN:wartość**: usuwa **wartość** z listy **dane** (jeśli istnieje).
- **POKAZ**: wyświetla aktualną zawartość listy **dane**.

d. Korutyna powinna być resetowalna za pomocą wyjątku **ResetProcesora**.

Zadanie do wykonania:

a. Stwórz dekorator **@korutyna** do automatycznego "primingu".

b. Stwórz wyjątek **ResetProcesora(Exception)**.

c. Zaimplementuj korutynę **procesor_polecen()**.

- W pętli while True użyj **try...except** do obsługi resetu.
- W bloku **try** odbieraj polecenia (**polecenie = yield**).
- Przetwarzaj polecenia: rozdzielaj string, używaj **if/elif/else** do obsługi akcji.

d. Przetestuj działanie:

- Stwórz instancję procesora.
- Wyślij kilka poleceń **DODAJ**.
- Wyślij polecenie **POKAZ**.
- Wyślij polecenie **USUN**.
- Zresetuj procesor za pomocą **.throw(ResetProcesora)**.
- Wyślij polecenie **POKAZ**, aby sprawdzić, czy dane zostały wyczyszczone.

Omówienie Zadania 4 - Procesor Poleceń

```
from functools import wraps

def korutyna(func):
    @wraps(func)
    def primer(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)
        return gen
    return primer

class ResetProcesora(Exception):
    """Wyjątek używany do resetowania procesora poleceń."""
    pass

@korutyna
def procesor_polecen():
    """Korutyna działająca jak prosty procesor poleceń."""
    print("--- Procesor poleceń gotowy. ---")
    dane = []
    while True:
        try:
            polecenie = yield

            # Parsowanie polecenia
            czesci = polecenie.strip().split(':', 1)
            akcja = czesci[0].upper()
            wartosc = czesci[1] if len(czesci) > 1 else None

            if akcja == 'DODAJ':
                dane.append(wartosc)
                print(f"DODANO: '{wartosc}'")
            elif akcja == 'USUN':
                if wartosc in dane:
                    dane.remove(wartosc)
                    print(f"USUNIĘTO: '{wartosc}'")
                else:
                    print(f"BŁĄD: Nie można usunąć, brak '{wartosc}'")
            elif akcja == 'POKAZ':
                print(f"AKTUALNE DANE: {dane}")
            else:
                print(f"BŁĄD: Nieznana akcja '{akcja}'")

        except ResetProcesora:
            print("--- RESETOWANIE PROCESORA ---")
            dane = []
        except Exception as e:
            print(f"Wystąpił nieoczekiwany błąd: {e}")

procesor = procesor_polecen()

procesor.send("DODAJ:jabłko")
procesor.send("DODAJ:banan")
procesor.send("POKAZ")
procesor.send("USUN:gruszka") # Błąd
procesor.send("USUN:jabłko")
procesor.send("POKAZ")

print("\n>>> Resetuję stan procesora...")
procesor.throw(ResetProcesora)
procesor.send("POKAZ") # Powinno być puste
```

Zadanie 5 - Budowa Własnych Iteratorów (Protokół Iteratora)

Problem: Używaliśmy wielu iteratorów (range, itertools, generatory). Ale jak sprawić, by nasza własna klasa była iterowalna? Jak działa for x in moj_obiekt:?

Rozwiązanie: Protokół Iteratora

- **Iterable (Iterowalny):** Obiekt, który można umieścić w pętli **for**. Musi mieć metodę **_iter_().**
- **Iterator:** Obiekt, który produkuje kolejne wartości. Musi mieć metodę **_next_()**, która rzuca **StopIteration** na końcu.

Pythoniczny Wzorzec: Najprostszym sposobem na uczynienie klasy iterowalną jest zaimplementowanie **_iter_** jako generatora.

Zadanie: Stwórz klasę **Odliczanie**, która będzie iterowalnym odlicznikiem.

a. Stwórz klasę **Odliczanie**.

b. W **_init_(self, start)** zapisz wartość początkową.

c. Zaimplementuj metodę **_iter_(self):**

- Wewnątrz tej metody użyj pętli **while**, aby **yield**-ować kolejne liczby od **self.start** w dół do 1.
- Po pętli, **yield**-uj finałowy komunikat, np. "START!".

d. Przetestuj:

- Stwórz instancję **odliczanie_do_startu** = **Odliczanie(5)**.
- Użyj pętli **for**, aby przeiterować po obiekcie i wyświetlić każdą wartość.

Omówienie Zadania 5 - Budowa Własnych Iteratorów

Cel: Analiza rozwiązania i zrozumienie, jak **_iter_** i **yield** tworzą "pythoniczny" sposób na uczynienie klasy iterowalną.

Kluczowe wnioski:

- Pętla **for** automatycznie wywołuje metodę **_iter_** na obiekcie.
- Nasza metoda **_iter_** zwraca **nowy obiekt generatora.**
- Pętla **for** następnie "konsumuje" ten generator, wywołując na nim **next()** aż do StopIteration.
- To elegancki wzorzec, który pozwala uniknąć pisania osobnej klasy iteratora z metodą **_next_**.

```
class Odliczanie:
    """Iterowalna klasa do odliczania od podanej liczby."""
    def __init__(self, start: int):
        self.start = start
        print(f"Utworzono obiekt Odliczanie(start={self.start})")

    def __iter__(self):
        """Zwraca iterator (w tym przypadku generator).
        Metoda jest wywoływana RAZ na początku pętli for.
        """
        print(">>> Pętla for wywołała __iter__ - tworzę generator!")
        liczba = self.start
        while liczba > 0:
            yield liczba
            liczba -= 1
        yield "START!"
        print(">>> Generator zakończył pracę.")

# --- Użycie ---
odliczanie_do_startu = Odliczanie(5)

print("\nZaczynam pierwszą pętlę for...")
for krok in odliczanie_do_startu:
    print(f"  Otrzymano z generatora: {krok}")

print("\nZaczynam drugą pętlę for (na tym samym obiekcie)...")
# Każda pętla for tworzy NOWY, świeży generator!
for krok in odliczanie_do_startu:
    print(f"  Otrzymano z generatora: {krok}")
```

Zadanie 6 - Pułapka Iteratora: Obiekt Jednorazowego Użytku

Problem: Nasza klasa **Odliczanie** była świetna, bo **_iter_** za każdym razem tworzył nowy generator. A co, jeśli zaimplementujemy **_iter_** i **_next_** ręcznie **w tej samej klasie**?

Antywzorzec (Pułapka): Obiekt, który jest swoim własnym iteratorem.

- Klasa implementuje zarówno _iter_, jak i _next_.
- Metoda _iter_ po prostu zwraca self.
- Stan (np. licznik) jest przechowywany bezpośrednio w obiekcie.

Zadanie (Eksperyment):

a. Stwórz klasę **LicznikJednorazowy**, która implementuje **zarówno _iter_**, jak i **_next_**.

b. W **_init_(self, max_val)** zapisz **self.max = max_val** i **self.n = 0**.

c. W **_iter_(self)**: po prostu return **self**.

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

- Jeśli **self.n < self.max**, zwiększ **self.n** o 1 i zwróć poprzednią wartość **self.n**.
- W przeciwnym razie, podnieś wyjątek **StopIteration**.

e. Przetestuj:

- Stwórz instancję **licznik = LicznikJednorazowy(3)**.
- Użyj pętli **for**, aby przeiterować po obiekcie.
- **Spróbuj przeiterować po nim drugi raz.** Co się stanie? Dlaczego?

Omówienie Zadania 6 - Pułapka Iteratora

Cel: Analiza wyników eksperymentu i zrozumienie, dlaczego iterator jest obiektem "jednorazowego użytku".

Wyjaśnienie:

- Obiekt **licznik** jest **swoim własnym iteratorem**.
- Pierwsza pętla **for** "konsumuje" iterator, wywołując **_next_** aż do momentu, gdy **self.n** osiągnie wartość **3**.
- Po zakończeniu pierwszej pętli, obiekt **licznik** jest w stanie wyczerpanym (**self.n** $==$ **3**).
- Druga pętla **for** prosi ten sam, **wyczerpany już obiekt** o iterator (**_iter_** zwraca **self**).
- Gdy druga pętla próbuje pobrać pierwszy element (**_next_**), metoda od razu rzuca **StopIteration**, ponieważ warunek **self.n** $<$ **self.max** jest fałszywy. Pętla natychmiast się kończy.

Wniosek: To jest fundamentalna różnica. **Iterator** jest stanowy i jednorazowy. **Obiekt iterowalny** (jak lista lub nasza klasa **Odliczanie**) jest "fabryką", która za każdym razem tworzy **nowy, świeży iterator**.

```
class LicznikJednorazowy:
    """ANTYWZORZEC: Iterator, który jest sam dla
siebie iteratorem.
    Można go zużyć tylko raz."""
    def __init__(self, max_val):
        self.max = max_val
        self.n = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.n < self.max:
            wynik = self.n
            self.n += 1
            return wynik
        else:
            raise StopIteration

licznik = LicznikJednorazowy(3)

print("Pierwsza pętla for:")
for liczba in licznik: print(f"  {liczba}")
# Wynik: 0, 1, 2

print("\nDruga pętla for (na tym samym obiekcie):")
for liczba in licznik: print(f"  {liczba}")
# Wynik: (brak)
```

Podsumowanie - Od Leniwych Strumieni do Interaktywnych Korutyn

Co Osiągnęliśmy w Module 6?

- Opanowaliśmy **zaawansowane narzędzia itertools** do "leniwego" przetwarzania danych (**groupby**, **tee**, **chain**, **islice**, **takewhile**...).
- Zrozumieliśmy, jak **generatory** ewoluują od prostych producentów danych do interaktywnych **korutyn**.
- Pogłębiliśmy wiedzę o **korutynach** (obsługa błędów **.throw(),** wzorzec dekoratora).
- Zgłębiliśmy **Protokół Iteratora** (**__iter__, __next__**) i nauczyliśmy się budować własne, iterowalne klasy.
- Zrozumieliśmy kluczową różnicę między **obiektem iterowalnym** (fabryką) a **iteratorem** (obiektem jednorazowego użytku).

Kluczowa Lekcja:

- Iteratory i generatory to fundament wydajnego przetwarzania danych w Pythonie.
- Korutyny oparte na generatorach to potężny wzorzec do budowy asynchronicznych, sterowanych zdarzeniami systemów.