# Iteratory, Generatory i Potoki Danych

**Gdzie jesteśmy?**

- Od zawsze używamy pętli **for.**
- Poznaliśmy **generatory (yield)** jako nowoczesny sposób tworzenia iteratorów.
- Odkryliśmy moduł **itertools** jako "szwajcarski scyzoryk" do pracy na sekwencjach.

**Nowe wyzwania:**

- Jak w praktyce zbudować własną klasę, po której można iterować?
- Jak przetwarzać gigantyczne pliki, nie wczytując ich do pamięci?
- Jak łączyć generatory i narzędzia **itertools** w wydajne **potoki przetwarzania danych?**

**Cel na dziś:** Przełożyć teorię z wykładu na praktyczne, wydajne i "pythoniczne" rozwiązania.

**Plan Działania:**

**Część 1: Protokół Iteratora w Praktyce**

- **Zadanie 1:** Budowa własnej klasy iteratora od zera **(WlasnyRange).**

**Część 2: Potęga Generatorów**

- **Zadanie 2:** Refaktoryzacja do generatora **(yield).**
- **Zadanie 3:** Praktyczny generator - "leniwe" wczytywanie pliku CSV.

**Część 3: Budowanie "Leniwych" Potoków Danych**

- **Zadanie 4:** Wyrażenia generatorowe w potoku przetwarzania.

**Część 4: Skrzynka z Narzędziami itertools**

- **Zadanie 5: itertools.groupby** w akcji - grupowanie danych.
- **Zadanie 6: itertools.tee** - rozdzielanie strumieni.

**Część 5: Kompletny Potok w Praktyce**

- **Zadanie 7:** Zadanie podsumowujące - analiza logów.

**Część 6: Zaawansowane Generatory (Korutyny)**

- **Zadanie 8: Dwukierunkowa komunikacja - .send() i .close().**

**Część 7: Podsumowanie**

# Zadanie 1 - Protokół Iteratora w Praktyce (WlasnyRange)

**Problem:** Jak zbudować własną klasę, która będzie działać z pętlą **for**, tak jak wbudowana funkcja **range()**?

**Rozwiązanie:** Musimy zaimplementować Protokół Iteratora.

- **Obiekt iterowalny (Iterable):** Klasa, która ma metodę **_iter_().** Ta metoda musi zwrócić **iterator**.
- **Iterator:** Klasa, która ma metodę **_next_()** (zwraca kolejny element lub rzuca **StopIteration**) oraz **_iter_()** (zwraca **self**).

**Zadanie:** Stwórz klasę **WlasnyRange**, która będzie iterowalna i będzie generować liczby od **start** do **stop** (bez **stop**).

**Stwórz klasę WlasnyRange (obiekt iterowalny):**

a. W konstruktorze **_init_(self, start, stop)** zapisz wartości start i stop.

b. Zaimplementuj metodę _iter_(self). Powinna tworzyć i zwracać instancję nowej klasy - WlasnyRangeIterator, przekazując jej self.

**Stwórz klasę WlasnyRangeIterator (iterator):**

a. W konstruktorze **_init_(self, wlasny_range_obj):**

- Zapisz referencję do obiektu **wlasny_range_obj**.
- Zainicjalizuj stan iteratora: **self.biezaca_wartosc** = **wlasny_range_obj.start.**

b. Zaimplementuj metodę **_iter_(self),** która po prostu zwraca **self.**

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

- Sprawdź, czy **self.biezaca_wartosc** jest mniejsza niż **stop** z obiektu **wlasny_range_obj**.
- Jeśli tak: zapamiętaj aktualną wartość, zwiększ **self.biezaca_wartosc** o 1 i zwróć zapamiętaną wartość.
- Jeśli nie: rzuć wyjątek **StopIteration**.

**Przetestuj:**

```
moj_zakres = WlasnyRange(2, 5)
for liczba in moj_zakres:
    print(liczba) # Powinno wyświetlić 2, 3, 4
```

# Omówienie Zadania 1 - Protokół Iteratora w Praktyce

**Cel:** Analiza rozwiązania i utrwalenie podziału ról na **Iterable** i **Iterator**.

**Kluczowe wnioski:**

- **WlasnyRange** to **fabryka iteratorów.** Jej jedyne zadanie to stworzenie obiektu **WlasnyRangeIterator** na żądanie pętli **for.**
- **WlasnyRangeIterator** to **wykonawca.** Przechowuje stan (**biezaca_wartosc)** i wykonuje logikę w **__next__.**
- **Problem:** To dużo kodu ("boilerplate") do napisania. Czy da się prościej?

# Zadanie 2 - Potęga Generatorów (yield)

**Problem:** Stworzyliśmy działający iterator, ale wymagało to dwóch klas i ręcznego zarządzania stanem (**biezaca_wartosc, StopIteration).**

**Rozwiązanie: Generator** - funkcja, która używa słowa kluczowego **yield.**

- Gdy metoda **_iter_** zawiera **yield**, automatycznie staje się **generatorem.**
- Python sam, "pod maską", tworzy dla nas obiekt iteratora, który zarządza stanem.
- **yield** "pauzuje" funkcję i zwraca wartość.
- Pętla **for** automatycznie obsłuży **StopIteration**, gdy generator zakończy działanie.

**Zadanie:** Zrefaktoryzuj klasę **WlasnyRange** do użycia generatora.

a. **Usuń klasę WlasnyRangeIterator** - nie będzie już potrzebna!

b. **Zmodyfikuj metodę _iter_ w klasie WlasnyRange:**
   - **Wewnątrz _iter_** stwórz zmienną, np. **biezaca_wartosc**, i ustaw ją na **self.start.**
   - Napisz pętlę **while**, która działa, dopóki **biezaca_wartosc** jest mniejsza niż **self.stop.**
   - Wewnątrz pętli, użyj **yield biezaca_wartosc**, aby "wyprodukować" kolejną liczbę.
   - Zwiększ **biezaca_wartosc** o 1.

c. **Przetestuj:** Upewnij się, że kod działa tak samo jak poprzednio.

# Omówienie Zadania 2 - Potęga Generatorów (yield)

**Cel:** Analiza rozwiązania z generatorem i podkreślenie jego zalet.

**Kluczowe wnioski:**

- **Zniknęła cała klasa** WlasnyRangeIterator!
- **Brak ręcznego zarządzania stanem:** Nie ma **self.biezaca_wartosc** w iteratorze, nie ma **__next__,** nie ma **raise StopIteration**.
- **Czytelność:** Logika jest teraz prostą pętlą **while**, a nie skomplikowaną maszyną stanu.
- **Problem:** Świetnie, umiemy generować proste sekwencje. A jak zastosować tę "leniwą" moc do realnych problemów, np. przetwarzania dużych plików?

# Zadanie 3 - Praktyczny Generator (Leniwe Wczytywanie Pliku CSV)

**Problem:** Jak przetworzyć duży plik CSV (np. 1GB), nie wczytując go całego do pamięci? Chcemy iterować po wierszach, które są już sparsowane.

**Rozwiązanie:** Stworzyć generator, który będzie czytał plik linia po linii, parsował każdą linię i **yield**-ował ją jako gotowy do użycia obiekt (np. słownik).

Narzędzia: Moduł **csv** i jego **csv.DictReader**.

**Zadanie:** Stwórz generator **czytaj_duzy_csv(sciezka_pliku)**, który będzie "leniwie" wczytywał dane z pliku CSV.

a. Przygotuj plik: Stwórz plik **dane.csv** z poniższą treścią:

```
imie,nazwisko,wiek
Anna,Kowalska,35
Piotr,Nowak,41
Zofia,Wisniewska,28
Jan,Jankowski,55
```

b. Napisz funkcję-generator: czytaj_duzy_csv(sciezka_pliku: str).

c. Wewnątrz funkcji, otwórz plik w bloku with.

d. Stwórz obiekt **csv.DictReader(plik)**. Automatycznie użyje on pierwszej linii jako nagłówków (kluczy słownika).

e. W pętli **for** iteruj po obiekcie **DictReader**.

f. Wewnątrz pętli, dla każdego wiersza (który jest słownikiem):

- Dokonaj prostej transformacji: zamień wartość pod kluczem '**wiek**' na **int**.
- Użyj **yield**, aby "wyprodukować" przetworzony wiersz (słownik).

g. Przetestuj: **Przeiteruj** po swoim generatorze, aby wyświetlić tylko osoby, których wiek jest powyżej 40 lat.

# Omówienie Zadania 3 - Praktyczny Generator

**Cel:** Analiza rozwiązania i podkreślenie wydajności pamięciowej.

**Kluczowe wnioski:**

- **Wydajność pamięciowa:** W pamięci znajduje się tylko jeden wiersz naraz. Możemy przetwarzać pliki o dowolnym rozmiarze.
- **Separacja odpowiedzialności:** Generator jest odpowiedzialny tylko za produkcję danych. Logika ich konsumpcji (filtrowanie, agregacja) jest na zewnątrz.
- **Problem:** Nasz kod testujący miesza iterację (**for**) z logiką (**if**). Czy możemy zbudować cały potok przetwarzania w bardziej elegancki, "pythoniczny" sposób?

# Zadanie 4 - Budowanie Potoków Danych (Wyrażenia Generatorowe)

**Problem:** Nasz kod **for...if** miesza produkcję danych z ich konsumpcją. Chcemy oddzielić te etapy, aby tworzyć reużywalne, "leniwe" potoki.

**Rozwiązanie:** Wyrażenia Generatorowe - składnia jak list comprehension, ale w nawiasach okrągłych **()**.

- **gen = (x*x for x in range(10) if x % 2 == 0)**
- Nie tworzą listy, lecz obiekt generatora.
- Można je łączyć w łańcuchy (potoki), gdzie każdy etap jest osobnym generatorem.

**Zadanie:** Zbuduj potok przetwarzania danych z pliku **dane.csv.**

- Użyj funkcji **czytaj_duzy_csv** z poprzedniego zadania jako źródła.
- **Etap 1 (Filtr):** Stwórz wyrażenie generatorowe **osoby_po_30**, które ze źródła weźmie tylko te osoby, których wiek jest większy niż 30.
- **Etap 2 (Transformacja):** Stwórz drugie wyrażenie generatorowe **opisy**, które weźmie dane z generatora **osoby_po_30** i dla każdej osoby stworzy string w formacie "**IMIE NAZWISKO**". Użyj metody **.upper()**.
- **Etap 3 (Konsumpcja):** Użyj pętli **for**, aby przeiterować po końcowym generatorze **opisy** i wyświetlić wyniki.

# Omówienie Zadania 4 - Budowanie Potoków Danych

**Cel:** Analiza rozwiązania i podkreślenie "leniwości" i kompozycyjności potoków.

Kluczowe wnioski:

- **Leniwość:** Cały potok jest "uśpiony", dopóki nie zaczniemy po nim iterować (np. pętlą **for**). Dane są przetwarzane jeden element na raz.
- **Kompozycyjność:** Każdy etap potoku to osobny, niezależny generator. Możemy je łatwo łączyć, zamieniać i reużywać.
- **Czytelność:** Kod jest deklaratywny. Opisujemy co ma się stać z danymi, a nie **jak** krok po kroku to zrobić w pętli.
- **Problem:** Potrafimy filtrować i transformować. A co, jeśli chcemy wykonać bardziej złożoną operację, np. **grupować** dane w strumieniu (np. wszystkie transakcje z tego samego dnia)?

# Zadanie 5 - Skrzynka z Narzędziami (itertools.groupby)

**Problem:** Jak grupować dane w strumieniu? Np. chcemy przetworzyć dane z pliku CSV i pogrupować osoby według pierwszej litery nazwiska.

**Rozwiązanie: itertools.groupby(iterable, key=...)** - potężne narzędzie do grupowania sąsiadujących elementów.

- Wymaganie: Dane wejściowe muszą być posortowane według tego samego klucza, po którym grupujemy!
- **groupby** zwraca iterator par: **(klucz_grupy, iterator_elementów_w_grupie)**.

**Zadanie:** Pogrupuj osoby z pliku **dane.csv** według pierwszej litery nazwiska i oblicz średni wiek w każdej grupie.

a. Użyj tego samego pliku **dane.csv** i generatora **czytaj_duzy_csv**.

b. **Etap 1 (Sortowanie): czytaj_duzy_csv** zwraca iterator. Nie możemy go posortować w miejscu. Musimy go najpierw "zmaterializować" do listy, a następnie posortować. Użyj **sorted(iterable, key=...)**. Kluczem będzie pierwsza litera nazwiska.

c. **Etap 2 (Grupowanie):** Użyj **itertools.groupby** na **posortowanej liście**. Kluczem znów będzie pierwsza litera nazwiska.

d. **Etap 3 (Agregacja):** W pętli **for** przeiteruj po wyniku z **groupby**. Dla każdej grupy:

   - Oblicz średni wiek osób w tej grupie. Pamiętaj, że **grupa** to też iterator, więc musisz go "zużyć".
   - Wyświetl literę grupy i obliczoną średnią.

# Omówienie Zadania 5 - itertools.groupby

**Cel:** Analiza rozwiązania, podkreślenie wzorca "sortuj-grupuj" i kompromisu "leniwy vs chciwy".

**Kluczowe wnioski:**

- **Wzorzec "Sortuj-Grupuj": groupby** wymaga posortowanych danych. To najważniejsza zasada.
- **Kompromis "Leniwy vs Chciwy":** Aby posortować, musieliśmy "zmaterializować" leniwy generator do listy (**sorted**), co zużyło pamięć. To częsty kompromis w przetwarzaniu danych.
- **Iterator w iteratorze: groupby** zwraca iterator, który sam produkuje kolejne iteratory (**grupa_iterator**).
- **Problem:** Świetnie. Ale co, jeśli chcemy przeprowadzić **dwie różne analizy** na tym samym strumieniu danych bez wczytywania pliku dwa razy?

# Zadanie 6 - Rozdzielanie Strumieni (itertools.tee)

**Problem:** Mamy jeden, "leniwy" generator (np. **czytaj_duzy_csv**). Chcemy przeprowadzić na jego danych **dwie niezależne analizy** (np. znaleźć max i policzyć średnią). Jak to zrobić, nie wczytując pliku dwa razy?

**Rozwiązanie: itertools.tee(iterable, n = 2)** - "rozdziela" jeden iterator na **n** niezależnych iteratorów.

- Gdy jeden z nowych iteratorów jest zużywany, **tee** buforuje elementy w pamięci, aby pozostałe iteratory mogły z nich skorzystać.
- **Uwaga:** Jeśli jeden iterator jest zużywany znacznie szybciej niż inne, bufor może urosnąć, zużywając pamięć.

**Zadanie:** Użyj **itertools.tee**, aby na podstawie jednego przejścia przez plik **dane.csv** wykonać dwie różne analizy.

a. Użyj generatora **czytaj_duzy_csv** jako źródła.

b. Użyj **itertools.tee**, aby stworzyć dwa niezależne iteratory: **iter_analiza1** i **iter_analiza2**.

**c.** **Analiza 1 (na iter_analiza1):** Znajdź osobę, której imię i nazwisko (połączone) mają największą długość. Użyj **max(iterable, key = ...)**.

**d.** **Analiza 2 (na iter_analiza2):** Oblicz, ile jest wszystkich osób i jaki jest ich łączny wiek.

e. Wyświetl wyniki obu analiz.

# Omówienie Zadania 6 - Rozdzielanie Strumieni (itertools.tee)

**Cel:** Analiza rozwiązania i zrozumienie, jak **tee** pozwala na wielokrotną, niezależną konsumpcję jednego strumienia.

**Kluczowe wnioski:**

- **Jedno źródło, wielu konsumentów: tee** pozwala na niezależne iterowanie po tym samym strumieniu danych bez ponownego wczytywania źródła (pliku).
- **Wewnętrzny bufor: tee** musi buforować elementy. Gdy jeden iterator wyprzedza drugi, bufor rośnie. To kompromis między I/O a pamięcią.
- **Kompletny warsztat:** Opanowaliśmy wszystkie kluczowe elementy do budowy "leniwych" potoków: tworzenie (generatory), filtrowanie/transformacja (wyrażenia generatorowe) i zaawansowane operacje (**itertools**).

# Zadanie 7 - Kompletny Potok Przetwarzania Danych (Podsumowanie)

**Problem:** Mamy duży plik z logami serwera. Chcemy znaleźć **3 adresy IP**, które wygenerowały **największy ruch** (sumę bajtów), ale tylko dla żądań zakończonych **błędem klienta (4xx).**

**Plan Działania (Kompletny Potok):** Połączymy wszystko, czego się nauczyliśmy.

a. **Źródło:** Generator **czytaj_logi(sciezka)** czytający plik linia po linii.

b. **Filtr 1:** Wyrażenie generatorowe filtrujące linie z kodem statusu **4xx**.

c. **Transformacja:** Wyrażenie generatorowe parsujące przefiltrowane linie i produkujące krotki **(ip, rozmiar_w_bajtach)**.

d. **Sortowanie (chciwe):** Zmaterializowanie i posortowanie krotek po adresie IP (wymóg **groupby**).

e. **Grupowanie i Agregacja (leniwe):** Użycie **itertools.groupby** do zsumowania ruchu dla każdego IP.

f. **Sortowanie końcowe i Wycięcie:** Posortowanie zagregowanych wyników po sumie ruchu i wzięcie 3 pierwszych.

**Zadanie:** Zaimplementuj powyższy potok, który obsłuży poniższy plik **logs.txt**

```
1.2.3.4 - - [11/Nov/2023] "GET /" 200 1500
5.6.7.8 - - [11/Nov/2023] "GET /admin" 401 100
1.2.3.4 - - [11/Nov/2023] "POST /login" 404 250
9.1.2.3 - - [11/Nov/2023] "GET /" 200 1800
5.6.7.8 - - [11/Nov/2023] "GET /data.json" 200 9500
1.2.3.4 - - [11/Nov/2023] "GET /static/img.jpg" 404 50
2.3.4.5 - - [11/Nov/2023] "GET /api/v1/users" 403 120
```

# Omówienie Zadania 7 - Kompletny Potok Przetwarzania Danych

**Cel:** Analiza kompletnego, wieloetapowego potoku i podsumowanie wszystkich technik.

**Kluczowe wnioski:**

- **Potęga kompozycji:** Połączyliśmy wiele małych, "leniwych" kroków w jeden potężny potok.
- **Świadomy kompromis:** Większość potoku jest "leniwa", ale zdecydowaliśmy się na "chciwe" sortowanie, bo było to konieczne dla **groupby**.
- **Deklaratywność:** Kod opisuje co chcemy osiągnąć, a nie **jak** to zrobić krok po kroku w zagnieżdżonych pętlach.

# Zadanie 8 - Zaawansowane Generatory (.send() i .close())

**Problem:** Do tej pory generatory tylko "wypluwały" dane. A co, jeśli chcemy sterować generatorem w trakcie jego działania lub przesyłać mu dane?

**Rozwiązanie:** Traktowanie generatora jak korutyny.

- **wartosc = yield wynik**: **yield** nie tylko zwraca **wynik**, ale też czeka na dane z zewnątrz, które trafią do zmiennej **wartosc**.
- **gen.send(dane):** Wznawia generator i przesyła mu **dane**.
- **gen.close():** Kończy działanie generatora (rzuca w nim **GeneratorExit**).

**Zadanie:** Stwórz "Inteligentny Czujnik" - generator obliczający średnią temperaturę, który można zresetować lub wyłączyć.

a. Napisz generator **monitor_temperatury(prog_alarmowy).**

b. Zainicjalizuj zmienne: **suma = 0**, **licznik = 0**, **srednia = 0**.

c. W nieskończonej pętli **while True**:

   - Użyj **odczyt = yield srednia**, aby zwrócić aktualną średnią i pobrać nowy odczyt.
   - Jeśli **odczyt** to **None** (np. wysłano pusty sygnał), zresetuj licznik i sumę (symulacja resetu urządzenia).
   - W przeciwnym razie: zaktualizuj sumę i licznik, oblicz nową średnią.
   - Jeśli średnia przekroczy **prog_alarmowy**, wypisz ostrzeżenie!

d. Obsłuż zamykanie: Owiń pętlę w blok **try...finally**. W bloku **finally** wypisz komunikat "Czujnik wyłączony".

e. **Użycie:**

   - Utwórz generator.
   - **Ważne:** Uruchom go pierwszy raz za pomocą **next(gen)** (tzw. priming).
   - Wyślij kilka temperatur za pomocą **.send()**.
   - Wyślij **None**, aby zresetować.
   - Zamknij generator za pomocą **.close().**

# Omówienie Zadania 8 - Zaawansowane Generatory

**Cel:** Zrozumienie dwukierunkowej komunikacji i cyklu życia generatora.

**Kluczowe wnioski:**

- **Korutyny:** Generatory mogą przechowywać stan i reagować na dane z zewnątrz. To fundament biblioteki asyncio.
- **Priming:** Zawsze pamiętaj o next() przed pierwszym .send().
- **Zarządzanie zasobami: try...finally** (lub obsługa **GeneratorExit**) pozwala bezpiecznie zamykać generatory.

# Podsumowanie - Od Protokołu Iteratora do Potoków Danych

**Co Osiągnęliśmy?**

- Zdemistyfikowaliśmy pętlę **for**, poznając **Protokół Iteratora (_iter_, _next_)**.
- Nauczyliśmy nasze własne obiekty, jak stać się iterowalnymi, najpierw "klasycznie", a potem nowocześnie za pomocą generatorów (yield).
- Zastosowaliśmy "leniwe" przetwarzanie do realnego problemu: analizy dużych plików bez zużywania pamięci.
- Opanowaliśmy budowę deklaratywnych **potoków danych** za pomocą **wyrażeń generatorowych**.
- Otworzyliśmy profesjonalną skrzynkę z narzędziami: **itertools** (**groupby**, **tee**) do zaawansowanych operacji na strumieniach.
- Poznaliśmy **dwukierunkową komunikację** z generatorami (**.send()**), co stanowi wstęp do programowania asynchronicznego.
- Połączyliśmy wszystkie te koncepcje, budując kompletny, wieloetapowy potok analityczny.

**Wielki Obraz: Zmiana Sposobu Myślenia**

1. Przeszliśmy od myślenia o danych w kategoriach "**chciwych**" **kolekcji** (listy) do myślenia w kategoriach "**leniwych**" **strumieni** (iteratory).
2. To pozwala pisać kod, który jest:
    - **Wydajny pamięciowo:** Przetwarza gigantyczne zbiory danych.
    - **Kompozycyjny:** Łączy proste kroki w złożoną logikę.
    - **Czytelny:** Opisuje co ma się stać, a nie jak.