Zaawansowane funkcje

Gdzie jesteśmy? Opanowaliśmy podstawy funkcji.

Cel na dziś: Uzupełnić wiedzę o zaawansowane, ale kluczowe techniki.

Agenda spotkania:

- **Zasięg zmiennych (Scope):** Zasada LEGB, **global** i **nonlocal.**
- **Ćwiczenie 1:** "Detektyw Zasięgu".
- **Rozpakowywanie kolekcji:** Przekazywanie ***listy** i ****słownika** do funkcji.
- **Ćwiczenie 2:** "Dynamiczny Raport".
- **Argumenty "keyword-only" (*):** Wymuszanie czytelności.
- **Dekoratory:** Anatomia i uniwersalny wzorzec.
- **Ćwiczenie 3:** Dekorator mierzący czas.
- **Generatory i yield:** Wydajne przetwarzanie danych.
- Zadania do samodzielnej pracy.

Ćwiczenie 1: “Detektyw Zasięgu” – Gdzie żyją zmienne?

Problem: Dlaczego zmienna zdefiniowana poza funkcją jest w niej widoczna, a zmienna zdefiniowana w funkcji nie jest widoczna na zewnątrz?

Rozwiązanie: Zasięg (Scope) - region programu, w którym dana nazwa (zmienna) jest dostępna.

Zasada LEGB: Kolejność, w jakiej Python szuka zmiennej:

Local: Wewnątrz bieżącej funkcji.

Enclosing: Wewnątrz funkcji zagnieżdżających.

Global: Na najwyższym poziomie modułu.

Built-in: W specjalnym module z wbudowanymi nazwami (print, len).

Ćwiczenie 1:

Cel: Przewidzenie wyniku działania programu bez jego uruchamiania, aby zrozumieć zasadę LEGB.

Zadanie: Przeanalizuj kod. Dla instrukcji **print** przewidź, co się wyświetli. Zapisz swoje przewidywania, a następnie uruchom kod i je zweryfikuj.

Omówienie Ćwiczenia 1: “Detektyw Zasięgu”

Cel: Analiza rozwiązania i utrwalenie zasady LEGB.

Kluczowa koncepcja: Python potrafi "patrzeć w górę" (od L do G), ale nie "patrzy w dół" ani "na boki".

Problem: Odczyt zmiennej globalnej działa. Ale co, jeśli spróbujemy ją zmodyfikować?

Modyfikacja zmiennych: global i nonlocal

Problem: Co się stanie, gdy spróbujemy przypisać nową wartość do zmiennej globalnej wewnątrz funkcji?

Rozwiązanie: Jawne zadeklarowanie intencji za pomocą słów kluczowych.

global nazwa_zmiennej: Mówi funkcji: "Nie twórz nowej, lokalnej zmiennej. Modyfikuj tę z zasięgu Globalnego."

nonlocal nazwa_zmiennej: Mówi funkcji: "Modyfikuj zmienną z najbliższego zasięgu Zagnieżdżającego (Enclosing)."

Ćwiczenie 2: Rozpakowywanie Kolekcji - "Odwrotna Magia"

Problem: Mamy dane w liście lub słowniku. Jak przekazać je do funkcji, która oczekuje wielu osobnych argumentów?

Rozwiązanie: Operatory rozpakowywania (* i **)

* **(gwiazdka):** Rozpakowuje iterowalny obiekt (listę, krotkę) na argumenty pozycyjne.

** **(dwie gwiazdki):** Rozpakowuje słownik na argumenty nazwane.

Cel: Zastosowanie rozpakowywania słownika do dynamicznego wywoływania funkcji.

Zadanie:

a. Skopiuj funkcję **generuj_raport** z poprzedniego laboratorium (tę z argumentami domyślnymi).

b. Stwórz słownik **dane_pracownika** zawierający niektóre dane, np. **{"imie": "Jan", "miasto": "Poznań"}.**

c. Wywołaj funkcję **generuj_raport**, rozpakowując ten słownik jako jej argumenty.

Omówienie Ćwiczenia 2: Rozpakowywanie Kolekcji

Cel: Analiza rozwiązania i utrwalenie wzorca rozpakowywania słownika.

Kluczowa koncepcja: ****słownik** przy wywołaniu funkcji rozpakowuje go na argumenty nazwane. Klucze w słowniku muszą pasować do nazw parametrów.

Argumenty "Keyword-Only" - Wymuszanie Czytelności

Problem: Mamy funkcję z wieloma argumentami, np. flagami **True/False.** Wywołanie **stworz_uzytkownika("Jan", True, False, True)** jest **nieczytelne**. Co oznaczają te wartości logiczne?

Rozwiązanie: Wymuszenie argumentów nazwanych za pomocą *

- Samotna gwiazdka * w definicji funkcji mówi: "Wszystkie parametry **po mnie** muszą być podane z nazwą".

Zadanie: Zdefiniuj funkcję **wyslij_email(adres, temat, tresc, *, priorytet="normalny").** Wywołaj ją, podając priorytet jako argument nazwany.

```
# Parametry po '*' muszą być podane jako nazwane
def stworz_uzytkownika(imie, *, jest_adminem,
                       aktywny):
    print(f"Tworzę użytkownika {imie}...")
    if jest_adminem:
        print("Nadaję uprawnienia admina.")
    if aktywny:
        print("Konto jest aktywne.")

# Poprawne wywołanie (czytelne!)
stworz_uzytkownika("Anna", jest_adminem=True,
                   aktywny=True)

# BŁĘDNE wywołanie - spowoduje TypeError
# stworz_uzytkownika("Piotr", True, False)
```

Argumenty "Keyword-Only" - Wymuszanie Czytelności

Cel: Analiza rozwiązania i utrwalenie zastosowania * do wymuszania argumentów nazwanych.

Kluczowa koncepcja: Samotna gwiazdka * działa jak bariera, zmuszając do użycia nazw dla wszystkich parametrów po niej.

Ćwiczenie 3: Dekoratory - "Owinięcie" Funkcji w Dodatkową Logikę

Problem: Jak dodać tę samą logikę (np. logowanie, mierzenie czasu) do wielu funkcji, nie zmieniając ich kodu (zasada DRY)?

Rozwiązanie: Dekorator - funkcja, która przyjmuje inną funkcję, "owija" ją w dodatkową logikę i zwraca nową, ulepszoną funkcję.

Zadanie: Stwórz prosty dekorator **@loguj**, który przed wywołaniem funkcji wyświetli "Start funkcji...", a po jej zakończeniu "Koniec funkcji.". Przetestuj go na prostej funkcji.

Omówienie Ćwiczenia 3: Prosty Dekorator Logujący

Cel: Analiza rozwiązania i utrwalenie podstawowej struktury dekoratora.

Kluczowa koncepcja: Wywołanie **przywitaj_swiat()** tak naprawdę uruchamia funkcję **wrapper**, która z kolei uruchamia oryginalną **przywitaj_swiat.**

Problem: Ten prosty dekorator działa tylko dla funkcji, które **nie przyjmują argumentów i niczego nie zwracają.**

Uniwersalny Wzorzec Dekoratora: Jak obsłużyć każdą funkcję?

Problem: Nasz prosty dekorator nie działa z funkcjami, które przyjmują argumenty lub zwracają wartości.

Rozwiązanie: Użycie *args i **kwargs do stworzenia uniwersalnego "przekaźnika".

Ćwiczenie 4 - "Uniwersalny Dekorator Mierzący Czas"

Cel: Zastosowanie uniwersalnego wzorca dekoratora do stworzenia użytecznego narzędzia.

Problem: Jak zmierzyć czas wykonania dowolnej funkcji, niezależnie od jej argumentów i zwracanej wartości?

Narzędzie: Moduł time.

- **time.time()** zwraca aktualny czas systemowy w sekundach (jako **float**).

Zadanie:

a. Stwórz dekorator **mierz_czas**, używając **uniwersalnego wzorca**.

b. Wewnątrz **wrappera**, **przed** wywołaniem oryginalnej funkcji, zapisz czas startu (time.time()).

c. Wywołaj oryginalną funkcję, przekazując jej ***args** i ****kwargs**, i **zapisz jej wynik** w zmiennej.

d. **Po** wywołaniu, zapisz czas końca i oblicz różnicę.

e. Wyświetl informację o czasie wykonania funkcji.

f. **Zwróć** oryginalny wynik, który zapisałeś w kroku 3.

g. Przetestuj dekorator na dwóch różnych funkcjach (przykłady poniżej).

Omówienie Ćwiczenia 4 (Dekorator Mierzący Czas)

Cel: Analiza rozwiązania i utrwalenie uniwersalnego wzorca dekoratora.

Ćwiczenie 5 - Generatory i yield: Leniwe Przetwarzanie Danych

Problem: Co, jeśli potrzebujemy przetworzyć milion liczb? Stworzenie milion-elementowej listy w pamięci jest **bardzo kosztowne**.

Rozwiązanie: Generator - specjalny rodzaj funkcji, która nie zwraca jednej wartości, ale produkuje **sekwencję wartości w locie**, jedną po drugiej, nie przechowując ich wszystkich w pamięci.

Kluczowe słowo: yield

- **yield** działa jak **return,** ale "pauzuje" funkcję, zapamiętując jej stan. Przy następnym wywołaniu, funkcja wznawia działanie od miejsca, w którym została spauzowana.

Cel: Stworzenie i użycie prostego generatora.

Zadanie:

- Zdefiniuj funkcję-generator **licz_do_trzech**.
- Wewnątrz funkcji, użyj **yield**, aby kolejno "zwrócić" liczby 1, 2 i 3.
- **Poza** funkcją, stwórz obiekt generatora.
- Użyj pętli **for**, aby przeiterować po generatorze i wyświetlić wszystkie wyprodukowane przez niego wartości.

Omówienie Ćwiczenia 5 (Prosty Generator)

Cel: Analiza rozwiązania i utrwalenie wzorca tworzenia i konsumowania generatora.

Zadanie 1: Generator Liczb Fibonacciego

Cel: Utrwalenie zaawansowanych technik pracy z funkcjami: dekoratorów, generatorów i ***args/****kwargs**.

Problem: Stwórz generator, który produkuje kolejne liczby z ciągu Fibonacciego (0, 1, 1, 2, 3, 5, 8...).

Kroki:

- Zdefiniuj funkcję-generator **fibonacci**, która przyjmuje jeden argument **limit** (ile liczb wygenerować).
- Wewnątrz użyj pętli, aby generować kolejne liczby ciągu za pomocą **yield**.
- Przetestuj, wyświetlając 10 pierwszych liczb ciągu.

Omówienie Zadania 1: Generator Liczb Fibonacciego

Cel: Analiza rozwiązania i utrwalenie wzorca generatora do tworzenia sekwencji.

Kluczowa koncepcja: Generator "pamięta" swój stan (a i b) pomiędzy kolejnymi wywołaniami yield.

Przetwarzanie Danych w Stylu Funkcyjnym

Gdzie jesteśmy? Opanowaliśmy zaawansowane mechanizmy funkcji.

Nowy problem: Często chcemy wykonać tę samą operację na całej kolekcji. Pisanie pętli for bywa rozwlekłe.

Rozwiązanie: Narzędzia do programowania w stylu funkcyjnym.

Agenda tej części:

a. **Funkcje anonimowe lambda:** Małe, jednorazowe funkcje.

b. **Transformacja danych za pomocą map():** Stosowanie funkcji do każdego elementu.

c. **Filtrowanie danych za pomocą filter():** Wybieranie elementów spełniających warunek.

d. **Ćwiczenie 5: Zastosowanie map** i **filter** w praktyce.

Narzędzia Stylu Funkcyjnego: lambda, map, filter

- **Funkcja lambda -** Anonimowa funkcja w jednej linii

  ```
  lambda argumenty: wyrażenie
  ```

  Cel: Tworzenie małych, jednorazowych funkcji bez potrzeby używania **def.**

- **Funkcja map()** - Transformacja kolekcji

  ```
  map(funkcja, sekwencja)
  ```

  Cel: Stosuje funkcję do każdego elementu sekwencji.

- **Funkcja filter()** - Selekcja z kolekcji

  ```
  filter(predykat, sekwencja)
  ```

  Cel: Zwraca tylko te elementy sekwencji, dla których predykat (funkcja zwracająca True/False) jest prawdziwy.

Przetwarzanie danych w stylu funkcyjnym

Cel: Zastosowanie map i filter do transformacji i selekcji danych.

Zadanie 1 (użycie map):

a. Stwórz listę **imiona = ["anna", "piotr", "zofia"].**

b. Użyj funkcji **map** i funkcji **lambda**, aby stworzyć nową listę, w której każde imię będzie miało pierwszą literę wielką (użyj metody **.capitalize()).**

c. Wynik działania **map** przekonwertuj na listę i zapisz w zmiennej **imiona_poprawione**.

d. Wyświetl nową listę, aby zweryfikować wynik.

Zadanie 2 (użycie filter):

a. Stwórz listę **oceny = [5, 3, 2, 5, 4, 2, 3, 4].**

b. Użyj funkcji **filter** i funkcji **lambda**, aby stworzyć nową listę, która będzie zawierać tylko oceny równe lub wyższe niż 4.

c. Wynik działania **filter** przekonwertuj na listę i zapisz w zmiennej **dobre_oceny**.

```
nowa_lista = list(map(lambda x: x.operacja(), stara_lista))
przefiltrowana_lista = list(filter(lambda x: x > warunek, stara_lista))
```

Omówienie zadań

Cel: Analiza rozwiązania i utrwalenie wzorców **map** + **lambda** oraz **filter** + **lambda**.

Kluczowa koncepcja: Zamiast pętli **for** i ręcznego budowania listy, opisaliśmy transformację w jednej, deklaratywnej linii.

Zadanie 2: "Dekorator sprawdzający typy"

Cel: Utrwalenie zaawansowanych technik pracy z funkcjami: dekoratorów, generatorów i ***args/****kwargs**.

Problem: Stwórz dekorator, który sprawdza, czy argumenty pozycyjne przekazane do funkcji są określonego typu.

Kroki:

a. Stwórz dekorator **@sprawdz_typy(typ_argumentu).** Zwróć uwagę, że to będzie dekorator przyjmujący argument!

b. Wewnątrz, **wrapper** powinien iterować po *args i sprawdzać, czy każdy z nich jest instancją **typ_argumentu** (użyj **isinstance()**).

c. Jeśli którykolwiek typ się nie zgadza, **wrapper** powinien wyświetlić komunikat błędu i zwrócić **None**, przerywając działanie.

d. Przetestuj go na funkcji **dodaj(a, b),** dekorując ją **@sprawdz_typy(int).**

Omówienie Zadania 2: "Dekorator sprawdzający typy"

Cel: Analiza zaawansowanego wzorca dekoratora przyjmującego argumenty.

Podsumowanie

Co dzisiaj osiągnęliśmy?

- Opanowaliśmy zaawansowane techniki pracy z funkcjami (**global**, **nonlocal**, *, **, **keyword-only**).
- Zbudowaliśmy uniwersalny, praktyczny **dekorator**.
- Nauczyliśmy się tworzyć i używać generatory do wydajnego przetwarzania danych.
- Poznaliśmy podstawy stylu funkcyjnego (**map, filter, lambda**).

Problem do rozwiązania: Nasz kod, choć potężny, wciąż jest w jednym pliku. Jak go profesjonalnie zorganizować i współdzielić?

Na następnych zajęciach (Laboratorium 6):

- **Moduły**: Tworzenie i importowanie własnych bibliotek.
- **Magia if __name__ == "__main__"**: Plik jako program i biblioteka.
- **Pakiety**: Grupowanie modułów w większe struktury.
- Dobre praktyki organizacji projektów w Pythonie.