Czysty kod i dobre praktyki

Gdzie jesteśmy? Potrafimy budować kompletne, wielomodułowe projekty i zarządzać ich zależnościami.

Cel: Nauczyć się, jak pisać kod, który jest nie tylko działający, ale też czytelny, łatwy w utrzymaniu i profesjonalny.

Agenda:

- **Rozgrzewka:** Najważniejsza pułapka Pythona - mutowalne argumenty domyślne.
- **Czysty Kod:** Dlaczego to takie ważne?
- **PEP 8 - Biblia Stylu Pythona:** Kluczowe zasady formatowania.
- **Ćwiczenie 2: "Linter"** - poprawiamy nieczytelny kod.
- **Docstringi i Komentarze:** Jak pisać użyteczną dokumentację.
- **Type Hinting (Podpowiedzi typów):** Nowoczesny standard w Pythonie.
- **Ćwiczenie 3:** Dodajemy podpowiedzi typów do naszego kodu.
- Zadania do samodzielnej pracy.

Ćwiczenie 1 (Rozgrzewka) - Najważniejsza Pułapka Pythona

Krok 1: Przewidywanie (NIE URUCHAMIAJ KODU!)

- Przeanalizuj poniższy kod. Jaki Twoim zdaniem będzie skład drugiej pizzy (Margherity)? Zapisz swoją odpowiedź.

```
def przygotuj_pizze(dodatki, baza=[]):
    # Każda pizza powinna mieć świeżą bazę: sos i ser
    baza.append("sos pomidorowy")
    baza.append("ser")
    baza.extend(dodatki)
    print(f"Pizza gotowa! Składniki: {baza}")

print("--- Zamówienie 1: Capricciosa ---")
przygotuj_pizze(["szynka", "pieczarki"])

print("\n--- Zamówienie 2: Margherita ---")
# Chcemy nową, prostą Margheritę (tylko sos i ser)
przygotuj_pizze([]) # Jaki będzie wynik?
```

Krok 2: Weryfikacja i Wyjaśnienie

- Przepisz i uruchom powyższy kod. Czy wynik Cię zaskoczył?
- Wyjaśnienie: Wartość domyślna baza = [] jest tworzona TYLKO RAZ, podczas definicji funkcji. Wszystkie wywołania dzielą tę samą listę bazową.

Zadanie:

a. Zrefaktoryzuj swoją funkcję przygotuj_pizze, używając pokazanego, poprawnego wzorca z None.

b. Uruchom kod ponownie i upewnij się, że teraz każde zamówienie tworzy nową, poprawną pizzę.

Omówienie Ćwiczenia 1 (Mutowalne Domyślne)

Cel: Analiza poprawnego rozwiązania i utrwalenie idiomu **parametr = None.**

Kluczowa koncepcja: Warunek **if baza is None:** jest sprawdzany przy każdym wywołaniu. Dzięki temu nowa lista **[]** jest tworzona za każdym razem, gdy funkcja jest wywoływana bez podania własnej bazy.

Co dalej? Skoro już wiemy, jak unikać tak podstępnych błędów, pora zająć się tym, jak pisać kod, który jest nie tylko poprawny, ale i **czytelny**.

Czysty Kod - Dlaczego to takie ważne?

"Każdy głupiec potrafi napisać kod, który zrozumie komputer. Dobry programista pisze kod, który zrozumieją ludzie." - Martin Fowler

Problem: Kod, który działa, ale jest nieczytelny, staje się:

- **Trudny w utrzymaniu:** Poprawianie błędów jest koszmarem.
- **Trudny w rozwijaniu:** Dodawanie nowych funkcji jest ryzykowne i powolne.
- **Źródłem błędów:** Niezrozumiały kod łatwo zepsuć.

Czysty Kod to kod, który jest:

- **Czytelny:** Łatwy do zrozumienia na pierwszy rzut oka.
- **Prosty:** Rozwiązuje problem w możliwie najprostszy sposób.
- **Spójny:** Trzyma się jednego, ustalonego stylu.
- **Przewidywalny:** Zachowuje się tak, jak się tego spodziewamy.

Złota zasada: Pisz kod tak, jakby osoba, która będzie go utrzymywać, była agresywnym psychopatą, który wie, gdzie mieszkasz.

Co dalej? Jak osiągnąć czystość? Zaczniemy od fundamentu: **PEP 8**, czyli oficjalnego przewodnika po stylu kodu w Pythonie.

PEP 8 - Biblia Stylu Pythona w Praktyce

PEP 8 to oficjalny dokument, który jest "dżentelmeńską umową" programistów Pythona, jak formatować kod.

Kluczowe zasady (wybrane):

- **Wcięcia:** 4 spacje na poziom.
- **Długość linii:** Maksymalnie 79-99 znaków.
- **Odstępy:** Spacja wokół operatorów (x = 1), po przecinku ([1, 2]).
- **Puste linie:** Dwie puste linie między funkcjami, jedna między metodami.
- **Nazewnictwo: snake_case** dla funkcji i zmiennych, **PascalCase** dla klas.

Narzędzia, które robią to za nas (a których można użyć na kolokwium):

- **Linter (flake8, pylint):** Analizuje kod i **wskazuje błędy** stylistyczne.
- **Formatter (black, autopep8):** Automatycznie poprawia formatowanie kodu.

Przykład "Przed" i "Po":

- **Źle (niezgodnie z PEP 8):**

```
def funkcja(arg1,arg2):
 x=arg1+arg2
 print('wynik to',x)
```

- **Dobrze (zgodnie z PEP 8):**

```
def funkcja(arg1, arg2):
    x = arg1 + arg2
    print(f"Wynik to: {x}")
```

Ćwiczenie 2: Detektyw i Architekt Kodu

Cel: Najpierw naprawić niedziałający kod, a następnie go ulepszyć, stosując zasady Czystego Kodu i PEP 8.

Zadanie: Poniższy kod **nie działa**. Zawiera błędy i jest źle zaprojektowany.

```
VAT=0.23
def print_receipt(data):
 total=0
 print("---PARAGON---")
 for item in data:
  price_with_tax=item['price']*(1+VAT)
  total+=price_with_tax
  print(f"{item['name']}... {price_with_tax}")
 print("SUMA: "+str(total))

# Przykładowe dane wejściowe
produkty = [
    {'nazwa': 'Mleko', 'cena_netto': 3.50},
    {'nazwa': 'Czekolada', 'cena_netto': 5.20}
]
print_receipt(produkty)
```

Twoje zadania:

a. **Detektyw:** Uruchom kod. Znajdź i napraw błędy, które uniemożliwiają jego działanie. Podpowiedź: Zwróć uwagę na typy danych i nazwy kluczy.

b. **Architekt:** Gdy kod już działa, popraw jego projekt i styl:

- Zmień nazwy zmiennych na bardziej opisowe **(data, item, total).**
- Unikaj zmiennych globalnych: przekaż **VAT** jako argument z wartością domyślną.
- Popraw formatowanie (spacje, puste linie).
- (Opcjonalnie) Sformatuj wyświetlaną cenę do dwóch miejsc po przecinku.

Omówienie Ćwiczenia 2: Od Błędów do Czystego Kodu

Cel: Analiza błędów, poprawionego kodu i wprowadzenie automatycznych narzędzi.

Problem: Ręczne formatowanie jest żmudne.

Rozwiązanie Profesjonalistów:

Automatyczny Formatter black

- **black** to "bezwzględny" formatter kodu. Nie dyskutuje, po prostu formatuje.
- **Instalacja (w aktywnym venv): pip install black**
- **Użycie (w terminalu): black nazwa_pliku.py**

```
def drukuj_paragon(lista_produktow, stawka_vat=0.23):
    """Drukuje sformatowany paragon dla listy produktów."""
    suma_brutto = 0
    print("--- PARAGON ---")

    for produkt in lista_produktow:
        # Używamy poprawnego klucza 'cena_netto'
        cena_brutto = produkt['cena_netto'] * (1 + stawka_vat)
        suma_brutto += cena_brutto
        # Używamy formatowania do 2 miejsc po przecinku i wyrównania
        print(f"{produkt['nazwa']:<15} | {cena_brutto:>8.2f} zł")

    print("-" * 26)
    print(f"{'SUMA:':<15} | {suma_brutto:>8.2f} zł")


# Przykładowe dane wejściowe z poprawnymi kluczami
produkty_do_paragonu = [
    {'nazwa': 'Mleko', 'cena_netto': 3.50},
    {'nazwa': 'Czekolada', 'cena_netto': 5.20}
]
drukuj_paragon(produkty_do_paragonu)
```

Ćwiczenie 3 - Listy Składane (List Comprehensions)

Problem: Pętle for do tworzenia list są rozwlekłe. map i filter bywają nieczytelne. Jak to zrobić lepiej?

Rozwiązanie: Lista Składana (List Comprehension) - kompaktowy i bardzo czytelny sposób na tworzenie list na podstawie innych kolekcji.

Anatomia Listy Składanej:

```
# Wersja z pętlą for
kwadraty = []
for x in range(10):
    kwadraty.append(x**2)

# Wersja z listą składaną
kwadraty_lc = [x**2 for x in range(10)]

# Wersja z warunkiem (odpowiednik filter + map)
parzyste_kwadraty = [x**2 for x in range(10) if x % 2 == 0]
```

Zadanie:

a. Mamy listę słowników **pracownicy**.

b. Użyj jednej listy składanej, aby stworzyć nową listę **lista_plac**.

c. Nowa lista ma zawierać **tylko pensje** (**pensja**) tych pracowników, którzy są na stanowisku "**Specjalista**".

```
pracownicy = [
    {"imie": "Anna", "stanowisko": "Specjalista", "pensja": 4500},
    {"imie": "Piotr", "stanowisko": "Manager", "pensja": 8000},
    {"imie": "Zofia", "stanowisko": "Specjalista", "pensja": 5200},
]

# Użyj listy składanej, aby wyciągnąć pensje tylko specjalistów
lista_plac = [] # Uzupełnij listę składaną

print(f"Lista płac dla specjalistów: {lista_plac}")
# Oczekiwany wynik: [4500, 5200]
```

Omówienie Ćwiczenia 3: "Pythoniczny" Styl

Cel: Analiza rozwiązania z użyciem listy składanej.

Ćwiczenie 4 - Pythoniczne Przetwarzanie Danych

Problem: Mamy zduplikowane dane. Jak je efektywnie odfiltrować i przetworzyć, używając idiomatycznych narzędzi Pythona?

Nowe Narzędzia:

- **Zbiór (set):** Kolekcja unikalnych, nieuporządkowanych elementów. Idealny do szybkiego sprawdzania, czy coś już widzieliśmy.
- **Lista Składana (List Comprehension):** Kompaktowy i czytelny sposób na tworzenie list. **[wyrazenie for element in sekwencja if warunek]**

Zadanie: "Deduplikator Danych"

Kroki:

a. Mamy listę słowników z powtarzającymi się danymi.

b. Stwórz pusty **zbiór widziane_id**, który będzie przechowywał ID już przetworzonych użytkowników.

c. Użyj **listy składanej**, aby stworzyć nową listę **unikalne_dane**.

d. Wewnątrz listy składanej, dla każdego **uzytkownika** z listy wejściowej, sprawdzaj, czy jego **id nie znajduje** się w zbiorze **widziane_id**.

e. Jeśli ID jest nowe, dodaj je do zbioru **widziane_id** i jednocześnie dodaj do nowej listy krotkę **(id, imie).**

```
zduplikowane_dane = [
    {'id': 1, 'imie': 'Anna'},
    {'id': 2, 'imie': 'Piotr'},
    {'id': 1, 'imie': 'Anna'}, # Duplikat
    {'id': 3, 'imie': 'Zofia'},
]

widziane_id = set()
unikalne_dane = [] # Uzupełnij listę składaną


print(unikalne_dane)
# Oczekiwany wynik: [(1, 'Anna'), (2, 'Piotr'), (3, 'Zofia')]
```

Ćwiczenie 5 - Pułapki Typów (Zagadka 1)

Problem: Python jest językiem dynamicznie typowanym, ale typy danych mają ogromne znaczenie. Ich interakcje mogą prowadzić do zaskakujących wyników.

Zadanie: Przeanalizuj poniższy fragment kodu. **Nie uruchamiając go,** spróbuj przewidzieć po ilu iteracjach pętla while zakończy działanie

```
licznik = 0
print(f"Start: {licznik}, typ: {type(licznik)}")

while licznik != 1.0:
    licznik += 0.1
    print(f"W pętli: {licznik}")

print(f"Koniec: {licznik}")
```

Ćwiczenie 5 - Pułapki Typów (Zagadka 2)

Problem: Chcemy stworzyć planszę do gry w kółko i krzyżyk. Wydaje się, że można to zrobić w jednej, sprytnej linijce...

Zadanie: Przeanalizuj poniższy kod. Nie uruchamiając go, spróbuj przewidzieć, jak będzie wyglądała plansza po postawieniu jednego 'X'.

```
# Chcemy stworzyć planszę do gry w kółko i krzyżyk 3x3
plansza = [['_'] * 3] * 3
print("Początkowa plansza:")
for wiersz in plansza:
    print(wiersz)

# Gracz stawia 'X' w lewym górnym rogu
print("\nGracz stawia 'X' w [0][0]...")
plansza[0][0] = 'X'

# Jak wygląda plansza TERAZ?
print("\nAktualna plansza:")
for wiersz in plansza:
    print(wiersz)
```

Ćwiczenie 6 - Podsumowanie: Mini-Analizator Danych

Problem: Mamy listę danych (np. oceny studentów). Jak napisać jedno, reużywalne narzędzie (funkcję), które przeanalizuje te dane i zwróci zwięzłe podsumowanie?

Cel: Połączenie wiedzy o funkcjach, listach, słownikach, pętlach i obliczeniach w jednym, praktycznym zadaniu.

Zadanie:

a. Napisz funkcję **analizuj_oceny**, która przyjmuje listę słowników. Każdy słownik reprezentuje studenta i ma klucze: "**imie**" (str) i "**ocena**" (int).

b. Wewnątrz funkcji, wykonaj następujące analizy:

- Oblicz **średnią ocenę** wszystkich studentów.
- Znajdź **najlepszego studenta** (cały słownik osoby z najwyższą oceną).
- Stwórz **zbiór wszystkich unikalnych ocen**, które wystąpiły.

c. Funkcja ma **zwrócić** słownik z wynikami, np. {"srednia": 4.5, "najlepszy_student": {...}, "unikalne_oceny": {3, 4, 5}}.

d. **Poza** funkcją, przygotuj przykładowe dane, wywołaj funkcję i ładnie wyświetl otrzymane podsumowanie.

Ćwiczenie 6 - Podsumowanie: Mini-Analizator Danych

Kod do uzupełnienia:

```
def analizuj_oceny(lista_studentow):
    """Analizuje listę ocen studentów i zwraca podsumowanie."""
    if not lista_studentow:
        return {"srednia": 0, "najlepszy_student": None,
        "unikalne_oceny": set()}

    # Uzupełnij logikę analizy...
    # Wskazówki:
    # - Użyj pętli for do iteracji.
    # - Użyj zmiennych do akumulacji sumy i śledzenia maksimum.
    # - Użyj zbioru do zebrania unikalnych ocen.

    return {} # Placeholder

# --- Główna część programu ---
dane_studentow = [
    {"imie": "Anna", "ocena": 5},
    {"imie": "Piotr", "ocena": 3},
    {"imie": "Zofia", "ocena": 5},
    {"imie": "Krzysztof", "ocena": 4},
]

# Wywołaj funkcję i wyświetl wyniki
# ...
```

Omówienie Ćwiczenia 6 - Mini-Analizator Danych

Cel: Analiza kompletnego rozwiązania i utrwalenie wzorców przetwarzania listy słowników.

Ćwiczenie 7: Pułapki Typów - Zagadka 3

Problem: Pobieramy klucze ze słownika. Co się stanie, jeśli zmodyfikujemy słownik po pobraniu kluczy?

Zadanie: Przeanalizuj poniższy kod. Nie uruchamiając go, spróbuj przewidzieć, co dokładnie wyświetli ostatnia instrukcja print.

```
kontakty = {"Anna": "123", "Piotr": "456"}
print(f"Początkowe kontakty: {kontakty}")

# Pobieramy "listę" kluczy
nazwy_kontaktow = kontakty.keys()
print(f"Pobrane nazwy: {nazwy_kontaktow}")

# Modyfikujemy ORYGINALNY słownik
print("\nDodajemy nowy kontakt 'Zofia'...")
kontakty["Zofia"] = "789"

# Jak teraz wyglądają "pobrane" wcześniej nazwy?
print("\nJak wyglądają nazwy_kontaktow TERAZ?")
print(list(nazwy_kontaktow))
```

Ćwiczenie 8 – Myślenie rekurencyjne: “Spłaszczanie” Listy

Problem: Mamy listę, która zawiera inne listy, a te z kolei mogą zawierać kolejne... Jak przetworzyć taką strukturę o nieznanej głębokości?

Rozwiązanie: Rekurencja (Rekursja) - funkcja, która wywołuje samą siebie, aby rozwiązać mniejszą wersję tego samego problemu.

Kluczowe elementy rekurencji:

a. **Przypadek bazowy (base case):** Warunek, który kończy rekurencję (np. gdy element nie jest już listą).

b. **Krok rekurencyjny (recursive step):** Wywołanie samej siebie na mniejszym podproblemie.

Cel: Napisanie funkcji, która przyjmuje zagnieżdżoną listę i zwraca nową, "płaską" listę zawierającą wszystkie elementy.

Kroki:

a. Napisz funkcję **splaszcz_liste**, która przyjmuje listę **elementy**.

b. Wewnątrz funkcji stwórz pustą listę na wyniki.

c. Przejdź pętlą **for** po wszystkich **elem** w liście **elementy**.

d. Dla każdego **elem** sprawdź, czy jest on listą (użyj **isinstance(elem, list)).**

e. **Jeśli jest listą (krok rekurencyjny):** Wywołaj **splaszcz_liste** na tym elemencie i dodaj zwrócone przez nią elementy do swojej listy wyników.

f. **Jeśli nie jest listą (przypadek bazowy):** Po prostu dodaj ten element do listy wyników.

g. Na końcu **zwróć** listę wyników.

```
def splaszcz_liste(elementy):
    splaszczona = []
    # Uzupełnij logikę rekurencyjną...

    return splaszczona

zagniezdzona_lista = [1, [2, 3], 4, [5, [6, 7]]]
wynik = splaszcz_liste(zagniezdzona_lista)
print(f"Lista zagnieżdżona: {zagniezdzona_lista}")
print(f"Lista spłaszczona: {wynik}")
# Oczekiwany wynik: [1, 2, 3, 4, 5, 6, 7]
```

Omówienie Ćwiczenia 8 - Myślenie Rekurencyjne

Cel: Analiza kompletnego rozwiązania i utrwalenie wzorca rekurencyjnego.

Przykładowe rozwiązanie (kod):

```
def splaszcz_liste(elementy):
    """Rekurencyjnie spłaszcza zagnieżdżoną listę."""
    splaszczona = []
    for elem in elementy:
        if isinstance(elem, list):
            # Krok rekurencyjny: jeśli element jest listą,
            # wywołaj na nim tę samą funkcję i dołącz jej wynik.
            splaszczona.extend(splaszcz_liste(elem))
        else:
            # Przypadek bazowy: jeśli element nie jest listą,
            # po prostu go dodaj.
            splaszczona.append(elem)
    return splaszczona

# --- Główna część programu ---
zagniezdzona_lista = [1, [2, 3], 4, [5, [6, 7]]]
wynik = splaszcz_liste(zagniezdzona_lista)
print(f"Lista spłaszczona: {wynik}")
# Wynik: [1, 2, 3, 4, 5, 6, 7]
```

Ćwiczenie 9 - Rekurencja w Słownikach: "Głębokie Wyszukiwanie"

Problem: Mamy złożony, zagnieżdżony słownik (np. z pliku JSON). Jak znaleźć wartość dla danego klucza, niezależnie od tego, jak głęboko jest on ukryty?

Rozwiązanie: Ponownie, rekurencja!

Logika rekurencyjna:

a. Przejdź pętlą po parach **klucz**, **wartosc** w bieżącym słowniku.

b. **Przypadek bazowy 1:** Jeśli **klucz** jest tym, którego szukamy**, zwróć wartosc**.

c. **Krok rekurencyjny:** Jeśli **wartosc** jest innym słownikiem, wywołaj na nim tę samą funkcję (zejdź "piętro niżej").

d. **Ważne:** Jeśli rekurencyjne wywołanie coś znalazło (zwróciło wartość inną niż None), **natychmiast zwróć ten wynik** "piętro wyżej".

e. **Przypadek bazowy 2:** Jeśli pętla się skończy i nic nie znaleziono na tym poziomie, zwróć None.

Zadanie: "Poszukiwacz Skarbów"

Cel: Napisanie funkcji, która rekurencyjnie przeszukuje zagnieżdżony słownik.

Kroki: Zaimplementuj funkcję **znajdz_wartosc(dane, szukany_klucz)** zgodnie z opisaną logiką.

Ćwiczenie 9 - Rekurencja w Słownikach: "Głębokie Wyszukiwanie"

Kod do uzupełnienia:

```
def znajdz_wartosc(dane, szukany_klucz):
    """Rekurencyjnie przeszukuje zagnieżdżony słownik w poszukiwaniu
    klucza."""
    # Uzupełnij logikę rekurencyjną...
    # Wskazówki:
    # - Użyj pętli for i .items() do iteracji po słowniku.
    # - Użyj isinstance(wartosc, dict) do sprawdzenia, czy wartość jest słownikiem.
    # - Pamiętaj o obsłudze wyniku z wywołania rekurencyjnego!
    return None # Zwróć None, jeśli nic nie znaleziono na tym poziomie

konfiguracja = {
    "uzytkownik": "admin",
    "baza_danych": {
        "host": "localhost",
        "port": 5432,
        "credentials": {
            "user": "db_user",
            "password": "secret_password" # To jest nasz "skarb"
        }
    }
}

haslo = znajdz_wartosc(konfiguracja, "password")
print(f"Znalezione hasło: {haslo}")
# Oczekiwany wynik: secret_password
```

Omówienie Ćwiczenia 9 - Rekurencja w Słownikach

Cel: Analiza kompletnego rozwiązania i utrwalenie wzorca rekurencyjnego przeszukiwania.

```
def znajdz_wartosc(dane, szukany_klucz):
    """Rekurencyjnie przeszukuje zagnieżdżony słownik w
    poszukiwaniu klucza."""
    for klucz, wartosc in dane.items():
        # Przypadek bazowy 1: Znaleziono na bieżącym poziomie
        if klucz == szukany_klucz:
            return wartosc

        # Krok rekurencyjny: Jeśli wartość jest słownikiem, "kop" głębiej
        if isinstance(wartosc, dict):
            wynik_z_glebi = znajdz_wartosc(wartosc, szukany_klucz)
            # Jeśli "kopanie" przyniosło efekt, natychmiast zwróć skarb
            if wynik_z_glebi is not None:
                return wynik_z_glebi

    # Przypadek bazowy 2: Przeszukano cały poziom i nic nie znaleziono
    return None
```

Podsumowanie

Co dzisiaj osiągnęliśmy?

- Odkryliśmy kluczowe pułapki języka: mutowalne argumenty, niedokładność **float**, referencje w listach i dynamiczne widoki w słownikach.
  Wniosek: Zrozumieliśmy różnicę między wartością a referencją.
- Przećwiczyliśmy zasady PEP 8 i refaktoryzację.
- Opanowaliśmy potężne, "pythoniczne" narzędzia jak **listy składane.**
  Wniosek: Nasz kod jest nie tylko działający, ale i czytelny.
- Nauczyliśmy się rozwiązywać problemy o nieznanej głębokości za pomocą rekurencji.
  Wniosek: Potrafimy radzić sobie z problemami, których nie da się rozwiązać prostą pętlą.

Gdzie jesteśmy? Przeszliśmy od pisania prostych skryptów do analizowania, refaktoryzowania i stosowania zaawansowanych wzorców projektowych.

Co dalej?

- Do tej pory mieliśmy dane (słowniki) i logikę (funkcje) osobno.
- Następny krok: **Programowanie Obiektowe (OOP).**
- Nauczymy się tworzyć własne typy danych (**class**), które łączą w sobie dane (atrybuty) i zachowania (metody).