

Jasne, zrozumiałem! Uproszczę konstrukcję i przedstawię każde zadanie w dwóch oddzielnych blokach kodu: pierwszy będzie zawierał samo zadanie wraz z niezbędnymi importami i definicjami (część do uzupełnienia), a drugi będzie kompletnym rozwiązaniem. Dzięki temu od razu widać, co jest do zrobienia, a co jest już gotowym rozwiązaniem.

Zadanie 1: Prosty Kalkulator z Funkcjami Wyższego Rzędu

Kod Zadania (do uzupełnienia)

```
import math
from typing import Callable, Optional

# Zadanie 1: Prosty Kalkulator z Funkcjami Wyższego Rzędu
# Cel: Stwórz kalkulator, który używa funkcji wyższego rzędu do definiowania operacji matematycznych.
# Zamiast czterech oddzielnych funkcji, będziemy mieli jedną funkcję 'calculate',
# która przyjmuje operację (inną funkcję) jako argument.
# Dodatkowo, obsłuż dzielenie przez zero, zwracając 'None' dla wskazania błędu.

class FunctionalCalculator:
    ZERO = 0.0

    def calculate(self, a: float, b: float, operation: Callable[[float, float], float]) -> float:
        """
        Wykonuje operację matematyczną na dwóch liczbach.
        :param a: Pierwsza liczba.
        :param b: Druga liczba.
        :param operation: Funkcja (callable), która definiuje operację (np. dodawanie, odejmowanie).
        :return: Wynik operacji.
        """
        # TODO 1: Zaimplementuj logikę wykonania operacji.
        pass

    def calculate_safe(self, a: float, b: float, operation: Callable[[float, float], Optional[float]]) -> Optional[float]:
        """
        Wykonuje operację matematyczną na dwóch liczbach, zwracając Optional[float]
        w celu bezpiecznej obsługi potencjalnych błędów (np. dzielenia przez zero).
        :param a: Pierwsza liczba.
        :param b: Druga liczba.
        :param operation: Funkcja (callable), która definiuje operację i zwraca Optional[float].
        :return: Wynik operacji lub None, jeśli wystąpił błąd.
        """
        # TODO 2: Zaimplementuj logikę bezpiecznego wykonania operacji.
        pass

    if __name__ == "__main__":
        calculator = FunctionalCalculator()

        # TODO 3: Zdefiniuj operacje dodawania, odejmowania i mnożenia jako funkcje lambda.
        add = None
        subtract = None
        multiply = None

        # TODO 4: Zdefiniuj funkcję dzielenia 'divide_safe', która zwraca None w przypadku dzielenia przez zero.
        def divide_safe(x: float, y: float) -> Optional[float]:
            pass
```

Zadanie 2: Analiza Statystyk Listy Liczb (Funkcyjne Podejście)

Kod Zadania (do uzupełnienia)

```
from functools import reduce
from typing import Dict, Optional, List

# Zadanie 2: Analiza Statystyk Listy Liczb (Funkcyjne Podejście)
# Cel: Zoptimalizuj kod do analizy listy liczb (min, max, średnia, suma),
# wykorzystując funkcję 'reduce' do wykonania wszystkich obliczeń w jednym przejściu.
# Odpowiedz na pytania w komentarzach.

def analyze_numbers_functional_single_pass(numbers: List[int]) -> Dict[str, Optional[float]]:
    """
    Analizuje listę liczb i zwraca statystyki (min, max, suma, średnia)
    wykorzystując funkcyjne podejście i pojedyncze przejście z reduce.
    """
    if not numbers:
        return {"min": None, "max": None, "sum": 0, "average": None}

    # TODO 1: Jakie korzyści płyną z pojedynczego przejścia po liście w porównaniu do wielu pętli?
    # Odpowiedź 1:

    # TODO 2: Zaimplementuj funkcję redukującą (akumulator), która będzie zbierać min, max, sumę i count.
    # Ta funkcja powinna przyjmować bieżący stan (słownik) i nową liczbę, zwracając nowy stan.
    def stats_accumulator(acc: Dict[str, float], x: int) -> Dict[str, float]:
        pass

    # TODO 3: Zdefiniuj początkową wartość akumulatora (initial_stats) dla funkcji reduce.
    # Pamiętaj o obsłudze przypadku, gdy 'numbers' nie jest pusta.
    initial_stats = None

    # TODO 4: Użyj 'reduce' z zaimplementowanym akumulatorem i wartością początkową.
    final_stats = None

    # TODO 5: Wyciągnij końcowe wartości statystyk i oblicz średnią, pamiętając o przypadku dzielenia przez zero.
    min_value = None
    max_value = None
    total_sum = None
    average_value = None

    return {
        "min": float(min_value) if min_value is not None else None,
        "max": float(max_value) if max_value is not None else None,
        "sum": float(total_sum),
        "average": average_value
    }
```

Zadanie 3: Filtrowanie i Mapowanie Listy Obiektów

Kod Zadania (do uzupełnienia)

```
from typing import Callable, List

# Zadanie 3: Filtrowanie i Mapowanie Listy Obiektów
# Cel: Mając listę obiektów 'Product', filtruj te, które spełniają warunek,
# a następnie przekształć je w inną formę, używając 'list comprehensions'.
# Zadbaj o niezmiennosć. Dodatkowo, wprowadź funkcję do tworzenia dynamicznych filtrów.

class Product:
    def __init__(self, name: str, price: float, in_stock: bool):
        self.name = name
        self.price = price
        self.in_stock = in_stock

    def __repr__(self):
        return f"Product(name='{self.name}', price={self.price}, in_stock={self.in_stock})"

def process_products(products: List[Product]) -> List[str]:
    """
    Filtruje produkty i zwraca listę ich nazw w formacie "Nazwa (Cena zł)"
    dla produktów dostępnych i droższych niż 50.
    """
    # TODO 1: Użyj list comprehension do przefiltrowania produktów,
    # które są "in_stock" (dostępne) ORAZ ich cena jest wyższa niż 50.0.
    # Następnie przekształć je w stringi w formacie "Nazwa (Cena zł)".
    # Wynik powinien być nową listą, nie modyfikując oryginalnej (zasada niezmienności).
    filtered_and_mapped_products = []
    return filtered_and_mapped_products

# TODO 2: Zaimplementuj funkcję 'create_price_filter', która przyjmuje minimalną cenę
# i zwraca inną funkcję, która będzie filtrować produkty na podstawie tej ceny.
def create_price_filter(min_price: float) -> Callable[[Product], bool]:
    pass

if __name__ == "__main__":
    products_list = [
        Product("Laptop", 1200.0, True),
        Product("Myszka", 25.0, True),
        Product("Klawiatura", 75.0, False),
        Product("Monitor", 300.0, True),
        Product("Kamera", 50.0, False),
        Product("Słuchawki", 120.0, True)
    ]
    # TODO 3: Wywołaj 'process_products' i wydrukuj wyniki.
    # processed_names = process_products(products_list)
    # print("Dostępne produkty (powyżej 50 zł):")

    # TODO 4: Użyj 'create price filter' do stworzenia dynamicznych filtrów
```

Zadanie 4: Kompozycja Funkcji do Przetwarzania Danych

Kod Zadania (do uzupełnienia)

```
from typing import Callable, List, Any

# Zadanie 4: Kompozycja Funkcji do Przetwarzania Danych
# Cel: Zaimplementuj funkcję 'compose', która przyjmuje wiele funkcji i zwraca jedną, złożoną funkcję,
# która je wywołuje w odpowiedniej kolejności (od prawej do lewej).
# Następnie zaimplementuj 'pipeline' dla kolejności od lewej do prawej.
# Użyj tych funkcji do przetworzenia listy danych.

def compose(*functions: Callable[[Any], Any]) -> Callable[[Any], Any]:
    """
    Tworzy złożoną funkcję, która wykonuje podane funkcje w kolejności
    od prawej do lewej (jak w matematyce: f(g(x))).
    :param functions: Funkcje do skomponowania.
    :return: Skomponowana funkcja.
    """
    # TODO 1: Zaimplementuj logikę kompozycji funkcji.
    pass

def pipeline(*functions: Callable[[Any], Any]) -> Callable[[Any], Any]:
    """
    Tworzy złożoną funkcję (pipeline), która wykonuje podane funkcje w kolejności
    od lewej do prawej.
    :param functions: Funkcje do skomponowania.
    :return: Skomponowana funkcja.
    """
    # TODO 2: Zaimplementuj logikę pipeline'u funkcji.
    pass

if __name__ == "__main__":
    # Przykładowe funkcje do kompozycji
    def add_five(x: int) -> int: return x + 5
    def multiply_by_two(x: int) -> int: return x * 2
    def to_string(x: int) -> str: return str(x) + "!"
    def capitalize_string(s: str) -> str: return s.upper()

    print("--- Kompozycja funkcji (f(g(x))) ---")
    # TODO 3: Użyj 'compose' do stworzenia funkcji, która najpierw doda 5,
    # potem pomnoży przez 2, potem zamieni na string, a na końcu zamieni na duże litery.
    # Pamiętaj o kolejności (od prawej do lewej dla compose).
    # transform_data_composed = compose(...)
    # print(f"Wynik compose dla 3: {transform_data_composed(3)}")

    print("\n--- Pipeline funkcji (f | g | h) ---")
    # TODO 4: Użyj 'pipeline' do stworzenia funkcji, która najpierw doda 5,
    # potem pomnoży przez 2, potem zamieni na string, a na końcu zamieni na duże litery.
    # Pamiętaj o kolejności (od lewej do prawej dla pipeline).
    # transform_data_pipeline = pipeline(...)
    # print(f"Wynik pipeline dla 3: {transform_data_pipeline(3)}")
```

Zadanie 5: Zaawansowane Przetwarzanie Kolekcji z `reduce` i Immutability

Kod Zadania (do uzupełnienia)

```
from functools import reduce
from typing import List, Dict, Any

# Zadanie 5: Zaawansowane Przetwarzanie Kolekcji z 'reduce' i Immutability
# Cel: Mając listę obiektów 'Transaction', użyj funkcji 'reduce' do zagregowania danych
# w bardziej złożony sposób, na przykład do obliczenia salda konta i kategoryzacji transakcji.
# Upewnij się, że proces jest niezmienny, tj. nie modyfikuje oryginalnych obiektów
# ani posrednich struktur danych akumulatora.

class Transaction:
    def __init__(self, type: str, amount: float):
        self.type = type # "deposit" or "withdrawal"
        self.amount = amount

    def __repr__(self):
        return f"Transaction(type='{self.type}', amount={self.amount})"

def calculate_balance_immutable(transactions: List[Transaction]) -> float:
    """
    Oblicza końcowe saldo konta na podstawie listy transakcji,
    zachowując niezmiennosć.
    """
    # TODO 1: Zdefiniuj funkcję redukującą (accumulator), która będzie przyjmować
    # aktualne saldo (accumulator) i pojedynczą transakcję.
    # Funkcja powinna zwracać nowe saldo.
    def accumulator(current_balance: float, transaction: Transaction) -> float:
        pass

    # TODO 2: Użyj `reduce` do zastosowania funkcji `accumulator` do listy transakcji.
    # Pamiętaj o początkowym saldzie (initial_value).
    initial_balance = 100.0
    final_balance = None
    return final_balance

def categorize_transactions_immutable(transactions: List[Transaction]) -> Dict[str, List[Transaction]]:
    """
    Kategoryzuje transakcje na depozyty i wypłaty, zwracając nową,
    niezmienną strukturę danych.
    """
    # TODO 3: Zdefiniuj funkcję redukującą, która będzie tworzyć słownik
    # z kategoryzowanymi transakcjami.
    # Pamiętaj o niezmienności: zawsze zwracaj nowy słownik/listę.
    def category_reducer(acc: Dict[str, List[Transaction]], transaction: Transaction) -> Dict[str, List[Transaction]]:
        # TODO 4: Wyjaśnij, dlaczego tutaj konieczne jest tworzenie NOWYCH kopii słownika i list.
        # Odpowiedź 4:
        pass

    # TODO 5: Zdefiniuj początkową wartość akumulatora (initial_categories).
    initial_categories = None

    # TODO 6: Użyj `reduce` do kategoryzacji transakcji.
    categorized_data = None
    return categorized_data

if __name__ == "__main__":
    transaction_list = [
        Transaction("deposit", 50.0),
        Transaction("withdrawal", 20.0),
    ]
```