

Programowanie funkcyjne

Kacper Ślęzak, Tymon Woźniak

Wprowadzenie: Co to jest programowanie funkcyjne?

Wyobraź sobie, że pisziesz instrukcję obsługi do jakiejś maszyny. Możesz opisać to na dwa sposoby:

- **Sposób 1 (imperatywny):** "Weź śrubokręt, odkręć śrubkę A, następnie podnieś klapkę B, a potem wciśnij przycisk C." To jest jak **programowanie imperatywne** – dajesz komputerowi dokładne instrukcje, krok po kroku, co ma zrobić i jak ma to zrobić, modyfikując przy tym różne stany (np. status klapki, czy śrubka jest odkręcona).
- **Sposób 2 (funkcyjny):** "Zapewnij, że klapka jest podniesiona, a maszyna wyłączona, aby uzyskać wynik." Tutaj opisujesz, **co** chcesz osiągnąć, a nie **jak** dokładnie to zrobić. Skupiasz się na **logice transformacji danych** – przekształcaniu jednego stanu w drugi za pomocą niezmiennych operacji. To jest bliższe **programowaniu funkcyjnemu (FP)**.

Co to jest programowanie funkcyjne (FP) w praktyce?

To **paradygmat programowania**, czyli pewien **sposób myślenia i podejścia do tworzenia oprogramowania**. W FP patrzymy na obliczenia jak na **ewaluację funkcji matematycznych**.

Myśl o tym tak: masz zestaw funkcji (jak w matematyce: $f(x)=x^2$ albo $g(x,y)=x+y$). Te funkcje przyjmują jakieś dane wejściowe i zwracają wynik, ale co najważniejsze – **niczego nie zmieniają na zewnątrz**. Nie modyfikują żadnych globalnych zmiennych, nie zmieniają stanu systemu, po prostu obliczają i zwracają to, co mają zwrócić.

Kluczowe cechy FP, które ułatwiają życie programistom:

- **Deklaratywność**: Zamiast mówić komputerowi *jak* coś zrobić (np. "przejdź przez listę, dodaj do zmiennej `suma`"), mówimy mu co chcemy osiągnąć (np. "oblicz sumę tej listy"). Kod staje się bardziej zwężły i łatwiejszy do zrozumienia, bo odzwierciedla intencje, a nie szczegóły implementacji.
- **Brak efektów ubocznych**: Funkcje w programowaniu funkcyjnym są jak "czarne skrzynki". Wrzucasz do nich dane, dostajesz wynik, i nic więcej się nie dzieje. Nie wpływają na otoczenie, nie zmieniają danych poza swoim zakresem. Dzięki temu łatwiej przewidzieć, jak zachowa się program, bo jedna funkcja nie zepsuje czegoś, co robi inna.

Kluczowe cechy FP cd.

- **Przewidywalność**: Skoro funkcje nie mają efektów ubocznych i zawsze zwracają ten sam wynik dla tych samych danych wejściowych, to zachowanie programu jest **dużo łatwiejsze do przewidzenia i debugowania**. Jeśli coś pójdzie nie tak, wiesz, że problem jest w samej funkcji, a nie w jakimś ukrytym stanie, który zmienił się gdzieś indziej.

💡 **Analogia**: Wyobraź sobie kalkulator. Wrzucasz " $2 + 2$ ", dostajesz "4". Niezależnie od tego, ile razy to zrobisz, wynik zawsze będzie ten sam, a kalkulator nie zmieni swojej "temperatury" ani nie uszkodzi się od tego działania. To jest właśnie esencja czystej funkcji i myślenia funkcyjnego!

Porównanie Paradygmatów

FP vs Imperatywne vs Obiektowe

Aspekt	Funkcyjne	Imperatywne	Obiektowe
Dane	Niezmiennie	Zmienne	Hermetyzowane
Kontrola	Funkcje	Sekwencja kroków	Metody obiektów
Stan	Unikany	Centralny	Enkapsulowany

Kiedy Czego Użyć?

- **Funkcyjne**: Przetwarzanie danych, transformacje, analiza.
- **Imperatywne**: Algorytmy krok-po-kroku, kontrola przepływu.
- **Obiektowe**: Modelowanie realnych systemów, duże aplikacje.

Czyste Funkcje: Fundament FP

Definicja

- Dla tych samych danych wejściowych **zawsze zwraca ten sam wynik.**
- **Nie powoduje żadnych efektów ubocznych.**

Charakterystyka

- **Determinizm**: Zawsze ten sam wynik.
- **Brak efektów ubocznych**: Bez modyfikacji danych globalnych.
- **Referencyjna przezroczystość**: Można zastąpić wywołanie funkcji jej wynikiem.

 CZYSTA FUNKCJA

```
def square(x):  
    return x * x
```

 NIECZYSTA FUNKCJA

```
counter = 0  
def impure_increment():  
    global counter  
    counter += 1 # Modyfikuje stan globalny!  
    return counter
```

Zalety Czystych Funkcji

- **Łatwiejsze testowanie**: Brak ukrytych zależności.
- **Możliwość memoizacji**: Cache'owanie wyników.
- **Równoległość**: Bezpieczne w środowiskach wielowątkowych.
- **Kompozycja**: Łatwe łączenie.

Czyste Funkcje a Obsługa Błędów

Jak czyste funkcje radzą sobie z błędami?

W tradycyjnym programowaniu błędy często sygnalizuje się poprzez:

- **Rzucanie wyjątków:** `raise ValueError("Dzielenie przez zero!")`
- **Modyfikowanie stanu zewnętrznego:** zapis do logów, flaga błędu.

W programowaniu funkcyjnym unikamy tego, ponieważ:

- Rzucanie wyjątków jest **efektem ubocznym** (zmienia przepływ programu w sposób nieprzewidywalny dla wywołującego).
- Modyfikowanie stanu zewnętrznego również jest efektem ubocznym.


Funkcyjne podejście do błędów: Zwracanie wartości

Zamiast rzucać wyjątki, czyste funkcje **zwracają wynik, który wskazuje na sukces lub błąd**.

Typowe wzorce:


- **Optional / Maybe** (**Python:** `Optional[T]` **lub** `None`): Zwracamy oczekiwany typ danych lub `None` (lub specjalny obiekt), gdy operacja się nie powiodła. To sprawia, że wywołujący jest zmuszony jawnie obsłużyć oba przypadki.
- **Either / Result** (**Python:** `Tuple[bool, Union[T, Error]]`): Zwracamy parę: `(True, wynik)` w przypadku sukcesu lub `(False, informacja_o_błędzie)` w przypadku błędu. Bardziej szczegółowe.

Funkcyjne podejście do błędów cd.

```
#  Czysta funkcja z obsługą błędów przez zwracanie Optional
from typing import Optional

def safe_divide(a: float, b: float) -> Optional[float]:
    if b == 0:
        return None # Wskazuje na błąd bez efektu ubocznego
    return a / b

result_ok = safe_divide(10, 2) # 5.0
result_error = safe_divide(5, 0) # None
```

 **Pamiętaj:** Dzięki zwracaniu `None` (lub innego wskaźnika błędu) funkcja `safe_divide` pozostaje **czysta** – zawsze zwraca przewidywalny wynik dla danych wejściowych, a nie powoduje "nieoczekiwanego" przerwania programu.

Niezmienność i Rekursja

Niezmienność (Immutability)

Zamiast modyfikować istniejące dane, **tworzymy nowe struktury**.

```
# ❌ Zmienność
original_list = [1, 2, 3]
original_list.append(4) # Modyfikuje!


# ✅ Niezmienność
original_list = [1, 2, 3]
new_list = original_list + [4] # Tworzy nową!
```

Niezmiennność w Praktyce: Złożone Struktury

Tworzenie nowych kopii zamiast modyfikacji na miejscu

Gdy pracujemy ze słownikami, listami zagnieżdżonymi lub obiektami, mutacja może być subtelna.

W programowaniu funkcyjnym, zamiast zmieniać istniejące struktury, **tworzymy ich nowe kopie z pożądanymi zmianami**.

 **Pamiętaj:** operator `**` dla słowników (rozpakowanie) i `+` dla list (łączenie) tworzą **nowe obiekty**, co jest kluczowe dla zachowania niezmienności.

Tworzenie nowych kopii zamiast modyfikacji na miejscu cd.

```
# ❌ Mutacja słownika i listy w nim
data = {'user': 'Anna', 'tags': ['python', 'fp']}
data['user'] = 'Ewa'           # Modyfikacja na miejscu
data['tags'].append('dev')     # Modyfikacja listy na miejscu!

# ✅ Niezmienność słownika i listy w nim
data_immutable = {'user': 'Anna', 'tags': ['python', 'fp']}

# Tworzenie nowej kopii słownika z nową wartością
new_data_user = {**data_immutable, 'user': 'Ewa'}

# Tworzenie nowej kopii listy i nowej kopii słownika
new_tags = data_immutable['tags'] + ['dev']
new_data_tags = {**data_immutable, 'tags': new_tags}

print(f"Oryginalna data: {data_immutable}")
print(f"Po zmianie użytkownika: {new_data_user}")
print(f"Po dodaniu tagu: {new_data_tags}")
```


Rekursja jako Alternatywa dla Pętli

Wzorzec:

1. **Przypadek bazowy** (warunek stopu).
2. **Wywołanie rekurencyjne** (funkcja wywołuje siebie z mniejszym problemem).

```
def factorial(n):  
    if n == 0: # Przypadek bazowy  
        return 1  
    return n * factorial(n - 1) # Wywołanie rekurencyjne
```

Kiedy Używać Rekursji?

- Struktury drzewiaste (katalogi, DOM).
- Problemy typu "podziel i zwyciężaj".
- Naturalne definicje matematyczne.

Funkcje Wyższego Rzędu i Lambda

Funkcje Wyższego Rzędu (Higher-Order Functions)

Funkcje, które:

- Przyjmują inne funkcje jako argumenty.
- Zwracają funkcje.

```
def apply_operation(numbers, operation):  
    return [operation(num) for num in numbers]  
  
def square(x): return x ** 2  
numbers = [1, 2, 3, 4]  
squared = apply_operation(numbers, square) # [1, 4, 9, 16]
```

Lambdy (Funkcje Anonimowe)

Krótkie, jednolinijkowe funkcje dla prostych operacji.

```
# Zamiast: def add_one(x): return x + 1
add_one = lambda x: x + 1


# Praktyczne użycie:
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers)) # [1, 4, 9, 16, 25]
```

Częściowe Stosowanie Funkcji (Partial Application)

Tworzenie bardziej wyspecjalizowanych funkcji

Częściowe stosowanie funkcji (ang. *partial application*) polega na tworzeniu nowej funkcji poprzez **wstępne zdefiniowanie niektórych argumentów** funkcji oryginalnej.


Jest to potężna technika do tworzenia bardziej elastycznych i reużywalnych funkcji, które są "dostosowane" do konkretnego kontekstu.

 **Kiedy używać?**: Gdy masz ogólną funkcję, a potrzebujesz wielu wariantów tej funkcji z predefiniowanymi parametrami. Zwiększa to czytelność i modułowość.

Tworzenie bardziej wyspecjalizowanych funkcji cd.

```
from typing import Callable

# Oryginalna funkcja (bardziej ogólna)
def filter_by_price(product_price: float, min_price: float) -> bool:
    return product_price > min_price

#  Częściowe stosowanie: tworzymy funkcję zwracającą funkcję
def create_price_filter(min_price: float) -> Callable[[float], bool]:
    # Zwracamy funkcję, która "pamięta" min_price
    return lambda product_price: product_price > min_price

# Tworzymy wyspecjalizowane filtry
is_expensive = create_price_filter(100.0) # Funkcja, która filtruje > 100
is_very_expensive = create_price_filter(500.0) # Funkcja, która filtruje > 500

print(f"Cena 120 zł jest droga? {is_expensive(120.0)}") # True
print(f"Cena 80 zł jest droga? {is_expensive(80.0)}")    # False
print(f"Cena 400 zł jest bardzo droga? {is_very_expensive(400.0)}") # False
```

Kluczowe Funkcje: `map()`, `filter()`, `reduce()`

Wizualizacja Przepływu Danych

Dane wejściowe: [1, 2, 3, 4, 5]



`map(x2)` → [1, 4, 9, 16, 25]



`filter(>10)` → [16, 25]



`reduce(sum)` → 41


Implementacje i Przykłady

```
from functools import reduce
data = [1, 2, 3, 4, 5]

# MAP: Transformacja każdego elementu
doubled = list(map(lambda x: x * 2, data)) # [2, 4, 6, 8, 10]

# FILTER: Selekcja elementów spełniających warunek
evens = list(filter(lambda x: x % 2 == 0, data)) # [2, 4]

# REDUCE: Agregacja do pojedynczej wartości
total = reduce(lambda acc, x: acc + x, data, 0) # 15
maximum = reduce(lambda acc, x: max(acc, x), data) # 5

#  REDUCE: Agregacja wielu statystyk w JEDNYM przejściu
# (Przykład bardziej złożony, ale pokazuje moc reduce)
def analyze_reducer(acc, x):
    return {
        'min': min(acc['min'], x),
        'max': max(acc['max'], x),
        'sum': acc['sum'] + x,
        'count': acc['count'] + 1
    }

initial_stats = {'min': float('inf'), 'max': float('-inf'), 'sum': 0, 'count': 0}
all_stats = reduce(analyze_reducer, data, initial_stats)
# all_stats = {'min': 1, 'max': 5, 'sum': 15, 'count': 5}
```


Praktyczne Zastosowania

- `map()`: Konwersja jednostek, formatowanie danych.
- `filter()`: Wyszukiwanie, walidacja danych.
- `reduce()`: Agregacje, statystyki, sumatory.

Kompozycja Funkcji

Budowanie Złożoności z Prostych Elementów

Matematycznie: $(f \circ g)(x) = f(g(x))$

```
def double(x): return x * 2
def increment(x): return x + 1
def square(x): return x ** 2

result = double(increment(square(3))) # 3 -> 9 -> 10 -> 20

# Funkcja kompozycji (od prawej do lewej)
def compose(*functions):
    def inner(arg):
        result = arg
        for func in reversed(functions): result = func(result)
        return result
    return inner

transform = compose(double, increment, square)
result = transform(3) # 20
```

Kompozycja Funkcji (Pipeline)

```
# Pipeline (od lewej do prawej)
def pipeline(data, *functions):
    result = data
    for func in functions: result = func(result)
    return result

result = pipeline(3, square, increment, double) # 3 -> 9 -> 10 -> 20
```

Korzyści Kompozycji

- **Modułowość**: Małe, testowalne funkcje.
- **Reużywalność**: Funkcje można łączyć na różne sposoby.
- **Czytelność**: Jasny przepływ transformacji danych.

Zastosowania w Praktyce

Programowanie funkcyjne to nie tylko teoria – to **praktyczne narzędzie**, które znajduje zastosowanie w wielu dziedzinach informatyki. Jego zasady, takie jak niezmienność i brak efektów ubocznych, sprawiają, że kod jest bardziej przewidywalny i łatwiejszy do utrzymania, co jest kluczowe w złożonych systemach.

Data Science

W świecie analizy danych, gdzie często operujemy na dużych zbiorach i wykonujemy na nich sekwencje transformacji, programowanie funkcyjne jest niezwykle przydatne.

Apache Spark: W rozproszonych systemach przetwarzania danych, takich jak Spark, programowanie funkcyjne jest fundamentem. Operacje na RDD (Resilient Distributed Datasets) czy DataFrame'ach (np. `map`, `filter`, `reduceByKey`) to w swej istocie **czyste funkcje**, które transformują dane bez modyfikowania oryginalnych zbiorów. To zapewnia **odporność na błędy** i **łatwe skalowanie** w klastrach.

Data Science cd.

Pandas: Biblioteka Pythona do manipulacji danymi, choć nie jest czysto funkcyjna, intensywnie wykorzystuje koncepcje FP. Metody takie jak `apply()`, `map()`, `filter()` czy `groupby()` pozwalają na **deklaratywne operacje** na całych kolumnach, zamiast pętli. Funkcje, które przekazujemy do tych metod, działają na pojedynczych elementach lub grupach, ale nie modyfikują oryginalnego obiektu DataFrame.

```
import pandas as pd
df = pd.DataFrame({'Produkt': ['Laptop', 'Mysz', 'Klawiatura'],
                  'Cena': [1200, 50, 150],
                  'Dostępny': [True, True, False]})

# Przykład mapowania: Obliczanie ceny netto dla każdego produktu
df['Cena_Netto'] = df['Cena'].apply(lambda cena: cena / 1.23) # Stosuje lambdę do każdej ceny
# Przykład filtrowania: Wybieranie tylko dostępnych produktów
df_dostępne = df[df['Dostępny'].apply(lambda x: x == True)]

print("DataFrame z ceną netto:\n", df)
print("\nDostępne produkty:\n", df_dostępne)
```

Uczenie Maszynowe

W uczeniu maszynowym, zwłaszcza w obszarze głębokiego uczenia, koncepcje funkcyjne są naturalnie wplecione w architekturę i działanie modeli.

TensorFlow/PyTorch: W bibliotekach do głębokiego uczenia neuronowe sieci są budowane jako **sekwencje warstw**, gdzie każda warstwa jest w zasadzie **funkcją**, która przyjmuje dane wejściowe (tensory) i zwraca przekształcone dane wyjściowe (inne tensory). Ten proces jest wysoce funkcyjny:

- **Kompozycja funkcji:** Model to kompozycja wielu warstw.
- **Niezmiennność tensorów:** Tensory są często **niezmienne**, operacje na nich zwracają nowe tensory, zamiast modyfikować istniejące.
- **Brak efektów ubocznych:** Warstwy operują na swoich wejściach i zwracają wyjścia, nie modyfikując stanu globalnego modelu w sposób nieprzewidziany.

Użycie

```
import tensorflow as tf
from tensorflow import keras

# Budowanie modelu jako sekwencji funkcji (warstw)
model = keras.Sequential([
    keras.layers.Dense(64, activation='relu', input_shape=(784,)), # Funkcja transformująca wejście
    keras.layers.Dense(32, activation='relu'),                     # Kolejna funkcja
    keras.layers.Dense(10, activation='softmax')                   # Funkcja końcowa
])

# Przepływ danych przez model to kompozycja tych funkcji
# output = model(input_data)
```

Tutaj `model` to de facto **skomponowana funkcja**, która bierze dane wejściowe i przekształca je przez serię niezależnych, czystych warstw.

Frontend Development

Współczesne frameworki frontendowe coraz mocniej czerpią z programowania funkcyjnego, zwłaszcza w zarządzaniu stanem i renderowaniu interfejsów.

React (JavaScript): React promuje użycie **komponentów funkcyjnych** (Functional Components), które są po prostu funkcjami przyjmującymi właściwości (props) i zwracającymi element UI. Są to **czyste funkcje**:

- Dla tych samych propsów zawsze zwracają ten sam UI.
- Nie powodują efektów ubocznych (chyba że użyjemy specjalnych `useEffect` hooków, które są mechanizmem do zarządzania efektami ubocznymi w kontrolowany sposób).

```
function WelcomeMessage(props) {  
  return <h1>Witaj, {props.name}!</h1>;  
}
```

Frontend Development cd.

Redux (JavaScript): Popularna biblioteka do zarządzania stanem w aplikacjach React. Opiera się na koncepcji **reduktorów (reducers)**, które są **czystymi funkcjami**. Reduktory przyjmują aktualny stan aplikacji i akcję, a następnie zwracają **nowy stan**, nie mutując oryginalnego. To zapewnia przewidywalność stanu i łatwość debugowania.

```
const counterReducer = (state = 0, action) => {  
  switch (action.type) {  
    case "INCREMENT":  
      return state + 1;  
    case "DECREMENT":  
      return state - 1;  
    default:  
      return state;  
  }  
};
```

Przetwarzanie Strumieni

Systemy do przetwarzania strumieniowego (tzw. "big data in motion") są idealnym scenariuszem dla programowania funkcyjnego.

Kafka Streams, Apache Flink, RxJS: Te technologie pozwalają na definiowanie transformacji i agregacji danych w sposób deklaratywny, bez przejmowania się niskopoziomowymi szczegółami zarządzania stanem czy przetwarzania równoległego. Operacje takie jak `map`, `filter`, `reduce`, `groupBy` są sercem tych systemów. Dane przepływają przez łańcuch czystych funkcji, co gwarantuje spójność i niezawodność przetwarzania w czasie rzeczywistym.

Programowanie funkcyjne oferuje solidne podstawy do budowania **niezawodnych, skalowalnych i łatwych w utrzymaniu** systemów, szczególnie tam, gdzie dane są przetwarzane i transformowane. Jego uniwersalne zasady przekraczają granice konkretnych języków i technologii.

Zalety i Wady

✓ Zalety

- **Jakość kodu:** Czytelność, łatwe testowanie i debugowanie.
- **Niezawodność:** Brak efektów ubocznych, niezmiennosc (brak race condition).
- **Wydajność:** Bezpieczna równoległość, memoizacja, optymalizacje kompilatora.

✗ Wady


- **Krzywa uczenia:** Zmiana sposobu myślenia, nowe koncepty.
- **Wydajność:** Więcej obiektów (pamięć, GC pressure), rekursja bez optymalizacji (stack overflow).
- **Praktyczność:** Niektóre algorytmy naturalnie imperatywne, trudna integracja z istniejącym kodem.


Dobre Praktyki

Jak Zacząć?

1. **Małe kroki:** `map/filter/reduce` zamiast pętli.
2. **Czyste funkcje:** Unikaj globalnego stanu.
3. **Kompozycja nad dziedziczeniem:** Łącz funkcje.
4. **Immutability first:** Twórz nowe obiekty.

Praktyczne Wskazówki

```
#  Dobrze - czysta funkcja
def calculate_tax(amount, rate): return amount * rate / 100

#  Źle - efekt uboczny
tax_total = 0
def calculate_tax_bad(amount, rate):
    global tax_total
    tax = amount * rate / 100
    tax_total += tax # Efekt uboczny!
    return tax
```


Praktyczne Wskazówki cd.

 Dobrze - kompozycja funkcji

```
def process_order(order):  
    return pipeline(  
        order,  
        validate_order,  
        calculate_total,  
        apply_discount,  
        add_tax  
    )
```

 Źle - długa funkcja z wieloma odpowiedzialnościami

```
def process_order_bad(order):  
    # 50 linii kodu robiących wszystko...  
    pass
```

Następne Kroki

Co Dalej?

- **Funkcyjne języki:** Haskell, Clojure, F#.
- **Biblioteki:** Ramda.js, toolz (Python), cats (Scala).
- **Wzorce:** Monady, funktory, aplikatywne funktory.
- **Projektowanie:** Event sourcing, CQRS.

Narzędzia do Eksperymentowania

- **Python:** `functools`, `itertools`, `operator`.
- **JavaScript:** Ramda, Lodash/FP, RxJS.
- **Online REPLs:** Repl.it, CodePen.

Podsumowanie

Programowanie Funkcyjne to potężny paradygmat, który:

- **Zwiększa jakość kodu** (czyste funkcje, niezmiennosc).
- **Ułatwia testowanie** (brak efektów ubocznych).
- **Wspiera równoległość** (bezpieczne współdzielenie danych).
- **Promuje kompozycję** zamiast dziedziczenia.

Pamiętaj: Użyj podejścia funkcyjnego tam, gdzie przynosi korzyści!