Testowanie i Jakość Oprogramowania

Akademia Tarnowska

Katedra Informatyki

Tomasz Gądek

Wykład 5: Elementy programowania funkcyjnego

Czym jest programowanie funkcyjne

- Programowanie funkcyjne to styl tworzenia programów, w którym patrzymy na obliczenia jak na rozwiązywanie równań matematycznych, unikając zmiennych stanowych i efektów ubocznych.
- Programowanie funkcyjne kładzie nacisk na wyrażenia i deklaracje, a nie na instrukcje. Jego celem jest pisanie bardziej przewidywalnego, łatwego do testowania i zrozumienia kodu.

Podejście imperatywne (przykład)

Opisuje **jak** osiągnąć wynik, zmienia stan programu krok po kroku.

```
nums = [1, 2, 3, 4]
squared_nums = []
for num in nums:
    squared_nums.append(num * num)
print(squared_nums)  # [1, 4, 9, 16]
```

Podejście deklaratywne (przykład)

Opisuje **co** ma zostać wykonane, bez określania kolejnych kroków.

```
nums = [1, 2, 3, 4]
squared_nums = list(map(lambda x: x * x, nums))
print(squared_nums)  # [1, 4, 9, 16]
```

Korzyści stosowania programowania funkcyjnego

- Prostsze testowanie i debugowanie (funkcje są zazwyczaj **czyste**).
- Brak efektów ubocznych (funkcje nie zmieniają stanu).
- Lepsza modularność (funkcje są niezależne).
- Czysty kod (funkcje wyższego rzędu i rekurencja).
- Niezmienność danych (niemutowalne dane).

Funkcja czysta

- Zawsze zwraca ten sam wynik dla tych samych danych wejściowych (deterministyczna).
- Nie ma efektów ubocznych (nie zmienia stanu zewnętrznego ani zmiennych globalnych).
- Nie modyfikuje parametrów wejściowych (argumenty są niezmienne).

Efekty uboczne

- Modyfikowanie zmiennych globalnych.
- Zapis do plików, baz danych lub wysyłanie danych w sieci.
- Wypisywanie wyników na ekranie.
- Modyfikowanie przekazanych argumentów.

Przykład funkcji czystej

```
def calculate_average(grades):
    total = sum(grades)
    count = len(grades)
    if count == 0:
        return 0
    return total / count
```

Funkcja wyższego rzędu

- Przyjmuje inną funkcję jako argument.
- Zwraca funkcję jako wynik.

Może również łączyć oba te działania.

Funkcja wyższego rzędu (funkcja jako argument)

```
def increment(x):
    return x + 1

def apply_function(func, value):
    return func(value)

print(apply_function(increment, 5))  # 6
```

Funkcja wyższego rzędu (funkcja zwraca funkcję)

```
def create_adder(n):
    def adder(x):
        return x + n
    return adder

add_five = create_adder(5)

print(add_five(10))  # 15
```

Funkcja jako obiekt pierwszej klasy

Oznacza, że funkcje w danym języku programowania mogą być traktowane jak inne obiekty pierwszej klasy (np. liczby, ciągi znaków), co oznacza, że:

- Funkcje mogą być przypisywane do zmiennych.
- Funkcje mogą być przekazywane jako argumenty do innych funkcji.
- Funkcje mogą być zwracane przez inne funkcje.

Funkcja anonimowa

Funkcja anonimowa (inaczej funkcja **lambda**) to funkcja, która:

- Nie ma przypisanej nazwy.
- Jest definiowana bez użycia słowa kluczowego `def`.
- Zwykle jest używana do prostych operacji, które można wyrazić w jednej linii.

Funkcja anonimowa (przykład)

```
add = lambda x, y: x + y

print(add(3, 5))  # 8
```

Lista składana

Lista składana (*ang. list comprehension*) to sposób tworzenia listy w Pythonie w jednej linii. Pozwala na:

- Tworzenie listy z elementów pochodzących z iterowalnego obiektu.
- Używanie warunków do filtrowania elementów.
- Wykonywanie operacji na elementach podczas ich dodawania do listy.

Lista składana (przykład)

```
even_squares = [x**2 for x in range(1, 21) if x % 2 == 0]

print(even_squares)  # [4, 16, 36, 64, 100, 144, 196, 256, 324, 400]
```

Partial Application

To technika, w której funkcja jest **częściowo zastosowana**, czyli część jej argumentów jest już znana (ustalona), a pozostałe będą podane później podczas wywołania funkcji.

Partial Application (przykład)

```
import functools

def divide(a, b):
    return a / b

divide_by_two = functools.partial(divide, b=2)

result = divide_by_two(10)
print(result)  # 5.0
```

Currying

- Currying umożliwia przekształcenie funkcji, która normalnie przyjmuje wiele argumentów, w szereg funkcji, które przyjmują po jednym argumencie.
- Funkcja dzieli jedno wywołanie funkcji na kilka etapów, gdzie każdy etap przyjmuje jeden argument, a wynik jest obliczany po przekazaniu wszystkich argumentów.

Currying (przykład)

```
def curried_price(price):
    def apply_discount(discount):
        def apply_tax(tax):
            return price * (1 - discount) * (1 + tax)
        return apply_tax
    return apply_discount

final_price = curried_price(100)(0.1)(0.2)  # 100 zł, 10% rabatu, 20% podatek
print(final_price)  # 108.0
```

Kompozycja funkcji

Kompozycja funkcji to sposób łączenia dwóch funkcji, gdzie wynik jednej funkcji jest automatycznie przekazywany jako argument do drugiej. To jak budowanie złożonego zadania z prostszych kroków.

Kompozycja funkcji (przykład)

```
def add_5(x):
    return x + 5

def multiply_2(x):
    return x * 2

def compose(f, g):
    return lambda x: f(g(x))

combined_function = compose(multiply_2, add_5)

result = combined_function(3)  # Najpierw 3 + 5 = 8, potem 8 * 2 = 16
print(result)  # 16
```

Genratory

Generatory w Pythonie to specjalny rodzaj funkcji, które zwracają wartości na żądanie zamiast przechowywać całą sekwencję w pamięci. Dzięki temu są bardziej efektywne pamięciowo niż zwykłe listy.

Generatory wykorzystują słowo kluczowe `yield`, które działa jak `return`, ale pozwala wstrzymać działanie funkcji i wznowić je później od tego samego miejsca.

Korzyści z używania generatorów: oszczędność pamięci, nie przechowują całej sekwencji w pamięci, tylko zwracają wartości na żądanie.

Generatory (przykład 1)

```
def phonebook():
    contacts = [
        "John Smith - 123456789",
        "Anna Johnson - 987654321",
        "Peter Williams - 567890123"]

    for contact in contacts:
        yield contact

book = phonebook()
print(next(book))
print(next(book))
print(next(book))
```

```
John Smith - 123456789
Anna Johnson - 987654321
Peter Williams - 567890123
```

Generatory (przykład 2)

```
def phonebook():
    contacts = [
        "John Smith - 123456789",
        "Anna Johnson - 987654321",
        "Peter Williams - 567890123"]

    for contact in contacts:
        yield contact

for book in phonebook():
    print(book)
```

```
John Smith - 123456789
Anna Johnson - 987654321
Peter Williams - 567890123
```

Domknięcie

Domknięcie (*ang. closure*) to funkcja, która zapamiętuje zmienne z otaczającego ją zakresu (nawet po zakończeniu jego działania). Dzięki temu może korzystać z tych wartości później, mimo że kontekst, w którym powstała już nie istnieje.

Domknięcie (przykład)

```
def counter():
    count = 0

    def increment():
        nonlocal count
        count += 1

        return count

    return increment  # returns the closure


click_counter = counter()

print(click_counter())  # 1
print(click_counter())  # 2
print(click_counter())  # 3
```

Rekurencja

Rekurencja (*łac. recurrere*) to technika programowania, w której funkcja wywołuje samą siebie, aż do osiągnięcia warunku zakończenia.

Rekurencja (przykład)

```
def sum_to(n):

    if n <= 0:
        return 0
    return sum_to(n - 1) + n

print(sum_to(3))
```

```
6
```

Rekurencja ogonowa

Rekurencja ogonowa (*ang. tail recursion)* to specjalny rodzaj rekurencji, w której wywołanie rekurencyjne jest ostatnią operacją w funkcji.

Rekurencja ogonowa (przykład)

```
def sum_to(n, acc = 0):

    if n <= 0:
        return acc
    return sum_to(n - 1, acc + n)

print(sum_to(3)) # 6
```

```
6
```

Dekorator

Dekorator w Pythonie to funkcja, która **modyfikuje** lub **rozszerza** funkcjonalność innej funkcji, klasy lub metody bez zmiany jej kodu źródłowego. Dekoratory są stosowane do **"owijania" funkcji** w celu dodania nowych możliwości (np. logowanie, sprawdzanie uprawnień, modyfikacja wyników).

Dekorator (przykład)

```
def log_decorator(func):
    def wrapper():
        print("Function is being called...")
        func()  # Wywołanie funkcji
        print("Function has been executed.")
    return wrapper

@log_decorator
def greet():
    print("Hello, world!")

# kiedy wywołasz greet(), faktycznie wywołujesz wrapper()
greet()
```

Dekorator (rezultat)

```
Function is being called...
Hello, world!
Function has been executed.
```

map()

`map()` to funkcja w Pythonie, która umożliwia zastosowanie funkcji do każdego elementu w iterowalnym obiekcie (np. liście), zwracając nową iterację z wynikami.

```
map(function, iterable)
```

- **function** – funkcja, która zostanie zastosowana do każdego elementu.
- **iterable** – obiekt, do którego funkcja ma zostać zastosowana.

map() (przykład)

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

filter()

`filter()` to funkcja w Pythonie, która filtruje elementy w iterowalnym obiekcie na podstawie funkcji warunkowej, zwracając tylko te elementy, które spełniają warunek.

```
filter(function, iterables)
```

- **function** – funkcja, która zwraca **True** lub **False** w zależności od warunku.
- **iterable** – obiekt, z którego elementy są filtrowane.

filter() (przykład)

```
numbers = [1, 2, 3, 4, 5]
filtered = list(filter(lambda x: x > 3, numbers))
print(filtered)  # [4, 5]
```

reduce()

`reduce()` to funkcja z modułu `functools`, która redukuje iterowalny obiekt do jednej wartości, stosując funkcję łączącą kolejne elementy.

```
from functools import reduce
reduce(function, iterable)
```

- **function** – funkcja, która łączy dwa elementy.
- **iterable** – obiekt, który jest przetwarzany przez funkcję.

reduce() (przykład)

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
sum_result = reduce(lambda x, y: x + y, numbers)
print(sum_result)  # 15
```

zip()

`zip()` to funkcja, która łączy dwa lub więcej iterowalnych obiektów (*np. list*) w jedną iterację, tworząc krotki, gdzie każdy element krotki pochodzi z tego samego indeksu w różnych iterowalnych obiektach.

```
zip(iterable1, iterable2, ...)
```

Jeśli iterowalne obiekty mają różną długość, `zip()` *kończy łączenie na najkrótszym obiekcie.*

zip() (przykład)

```
names = ["Alice", "Bob", "Charlie"]
scores = [85, 92, 88]

for zipp in zip(names, scores):
    print(zipp)

zipp_as_list = list(zip(names, scores))
print(zipp_as_list)
```

```
('Alice', 85)
('Bob', 92)
('Charlie', 88)
[('Alice', 85), ('Bob', 92), ('Charlie', 88)]
```

accumulate()

`accumulate()` to funkcja z modułu `itertools`, która wykonuje iteracyjną akumulację wartości w iterowalnym obiekcie, zwracając wynik dla każdego etapu akumulacji.

```
from itertools import accumulate
accumulate(iterable, function)
```

- **iterable** – obiekt, który jest akumulowany.
- **function** – funkcja (opcjonalna) akumulacji (domyślnie dodawanie).

accumulate() (przykład)

```
from itertools import accumulate

def mul(x, y):
    return x * y

data = [1, 2, 3, 4, 5]
result = list(accumulate(data, mul))
print(result)
```

```
[1, 2, 6, 24, 120]
```

sorted()

`sorted()` to wbudowana funkcja, która zwraca posortowaną wersję iterowalnego obiektu, nie modyfikując go bezpośrednio.

```
sorted(iterable, key=None, reverse=False)
```

- **iterable** – obiekt, który ma zostać posortowany.
- **key** – funkcja do określenia porządku sortowania (opcjonalne).
- **reverse** – czy sortowanie ma być w odwrotnej kolejności (opcjonalne).

sorted() (przykład)

```
numbers = [-5, 3, -1, 2, -4]
sorted_by_absolute_value = sorted(numbers, key=abs, reverse=True)
print(sorted_by_absolute_value)
```

Sortowanie liczb według ich wartości bezwzględnej (`abs()`).

```
[-5, -4, 3, 2, -1]
```

groupby()

`groupby()` to funkcja z modułu `itertools`, która grupuje kolejne identyczne elementy iterowalnego obiektu. Zwraca obiekt, który zawiera pary klucz-wartość, gdzie klucz to wartość grupy, a wartość to lista elementów w tej grupie.

```
groupby(iterable, key=None)
```

- **iterable** – obiekt, który ma zostać pogrupowany.
- **key** – opcjonalna funkcja, która określa, jak mają być grupowane elementy.

groupby() (przykład 1)

```
from itertools import groupby

words = ["apple", "bat", "bar", "atom", "book"]
first_letter = lambda word: word[0]

grouped_words = groupby(sorted(words), first_letter)
dict_words = {}

for key, val in grouped_words:
    dict_words[key] = list(val)

print(dict_words)
```

```
{'a': ['apple', 'atom'], 'b': ['bar', 'bat', 'book']}
```

groupby() (przykład 2)

```
from itertools import groupby

words = ["apple", "bat", "bar", "atom", "book"]
first_letter = lambda word: word[0]

grouped_words = groupby(sorted(words), first_letter)
dict_words = {key: list(val) for key, val in grouped_words}

print(dict_words)
```

```
{'a': ['apple', 'atom'], 'b': ['bar', 'bat', 'book']}
```

any() i all()

- `any()` zwraca **True**, jeśli przynajmniej jeden element w iterowalnym obiekcie jest prawdziwy. W przeciwnym wypadku zwraca **False**.
- `all()` zwraca **True**, jeśli wszystkie elementy w iterowalnym obiekcie są prawdziwe. W przeciwnym wypadku zwraca **False**.

```
any(iterable)
all(iterable)
```

- **iterable** – obiekt iterowalny (np. lista, krotka), który chcemy sprawdzić.

any() i all() (przykład)

```
values = [0, None, False, 5, "", 3]

any_result = any(values)
all_result = all(values)

print(f"any() result: {any_result}")
print(f"all() result: {all_result}")
```

```
any() result: True
all() result: False
```

Koniec