Функтори у функціональному програмуванні

Функтор (Functor) – це структура, яка підтримує операцію `map` (відображення) та зберігає контекст обчислення. Функтори дозволяють застосовувати функції до значень всередині контейнерів (списків, обгорток, об'єктів).

Функтор **зберігає структуру** та **трансформує значення** всередині.
Приклади: `list`, `Maybe`, `Future`, `Box`.
Використовується у **функціональному програмуванні для обробки даних**.

Функтор **не змінює структуру контейнера**, а лише трансформує його вміст.

Основні ідеї функторів

Контейнеризоване значення

Наприклад, список `List[int]`, обгортка `Maybe[int]` або `Future[str]`.

Можливість застосування функції до значення

Використовуючи `map`, можна модифікувати вміст без зміни контейнера.

Дотримання функторних законів

Ідентичність: `F(x).map(lambda v: v) == F(x)`.

Композиція: `F(x).map(f).map(g) == F(x).map(lambda v: g(f(v)))`.

Приклади функторів

Функтор на основі списку (list)

У Python **списки (`list`) – це приклад функторів**, тому що вони підтримують `map()`.

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

Тут `map()` застосовує `lambda x: x ** 2` до кожного елемента **без зміни структури списку**.

Реалізація власного функтора в Python

Створимо функтор `Box`, який обгортає значення і дозволяє застосовувати `map()`.

```
class Box:
    def __init__(self, value):
        self.value = value

    def map(self, func):
        return Box(func(self.value))

    def __repr__(self):
        return f"Box({self.value})"

# Використання функтора
box = Box(10)
new_box = box.map(lambda x: x * 2)
print(new_box)  # Box(20)
```

Функтор `Box` дозволяє застосовувати функції до значення всередині контейнера.

Функтор Maybe (для роботи з можливими None)

Функтор корисний для безпечної роботи з `None` значеннями. `Maybe` запобігає помилкам `NoneType` під час виклику функції.

```
class Maybe:
    def __init__(self, value):
        self.value = value

    def map(self, func):
        if self.value is None:
            return Maybe(None)
        return Maybe(func(self.value))

    def __repr__(self):
        return f"Maybe({self.value})"

# Використання Maybe-функтора
some_value = Maybe(5).map(lambda x: x * 10)
none_value = Maybe(None).map(lambda x: x * 10)
```

```
print(some_value)  # Maybe(50)
print(none_value)  # Maybe(None)
```

Функтор Future (для асинхронних обчислень)

Функтори можуть бути використані для роботи з асинхронними задачами. Функтор `Future` дозволяє застосовувати `map()`, не блокуючи виконання коду.

```
import concurrent.futures

class Future:
    def __init__(self, func):
        with concurrent.futures.ThreadPoolExecutor() as executor:
            self.future = executor.submit(func)

    def map(self, func):
        return Future(lambda: func(self.future.result()))

    def result(self):
        return self.future.result()

# Використання Future-функтора
future = Future(lambda: 10).map(lambda x: x * 2)
print(future.result())  # 20
```

Категорії функторів у функціональному програмуванні

Функтори бувають різних типів, і вони відіграють важливу роль у функціональному програмуванні, зокрема у категорній теорії та функціональних мовах (Haskell, Scala, F#).

Основні категорії функторів:

1. Коваріантні функтори (Covariant Functors)
2. Контраваріантні функтори (Contravariant Functors)
3. Аплікативні функтори (Applicative Functors)
4. Монади (Monads)
5. Функтори IO (I/O Functors)

Коваріантні функтори (Covariant Functors)

Це звичайний функтор, що відображає значення, не змінюючи контейнер. Вони дозволяють **застосовувати функцію до значення всередині контейнера** через `map()`.

Властивості коваріантного функтора:

- `F(x).map(id) == F(x)` **(ідентичність)**
- `F(x).map(f).map(g) == F(x).map(lambda v: g(f(v)))` **(композиція)**

Приклад коваріантного функтора `Box`:

```
class Box:
    def __init__(self, value):
        self.value = value

    def map(self, func):
        return Box(func(self.value))  # Трансформація значення, збереження контейнера

    def __repr__(self):
        return f"Box({self.value})"

box = Box(10).map(lambda x: x * 2)
print(box)  # Box(20)
```

Контраваріантні функтори (Contravariant Functors)

Зворотний функтор, де `map()` працює у зворотному напрямку.
Замість перетворення значення всередині контейнера, він перетворює функцію, яка отримує значення. Контраваріантні функтори застосовуються для роботи з логуванням, перетворенням вхідних даних.

Приклад контраваріантного функтора (`Contramap`):

```
class Printer:
    def __init__(self, func):
        self.func = func

    def contramap(self, preprocessor):
        return Printer(lambda x: self.func(preprocessor(x)))

    def print_value(self, value):
        print(self.func(value))
```

```
printer = Printer(lambda x: f"Value: {x}")

# Використання contramap для зміни поведінки функтора
string_printer = printer.contramap(str)
string_printer.print_value(100)  # "Value: 100"
```

Аплікативні функтори (Applicative Functors)

Функтори, що дозволяють застосовувати функції, які самі знаходяться у контейнерах. Використовують `apply()` (`ap()`) для комбінування ефектів. Аплікативні функтори корисні для роботи з ефектами (`Maybe`, `Future`).

Приклад аплікативного функтора `Maybe` (для обробки `None`):

```
class Maybe:
    def __init__(self, value):
        self.value = value

    def map(self, func):
        return Maybe(func(self.value)) if self.value is not None
else Maybe(None)

    def apply(self, func_container):
        return func_container.map(self.value) if self.value is not
None else Maybe(None)

    def __repr__(self):
        return f"Maybe({self.value})"

# Функція всередині Maybe
maybe_func = Maybe(lambda x: x * 2)

# Значення всередині Maybe
maybe_value = Maybe(10)

# Аплікативний виклик (функція застосовується до значення)
result = maybe_value.apply(maybe_func)
print(result)  # Maybe(20)
```

Монади (Monads)

Монада – це функтор, який підтримує `flatMap()` або `bind()`, що дозволяє "розгортати" вкладені контейнери. Це розширення аплікативних функторів, які дозволяють послідовність обчислень.

Монади дозволяють послідовні обчислення без if і вкладеності.

📌 Монада `Maybe` (обробка `None` без `if`):

```
class Maybe:
    def __init__(self, value):
        self.value = value

    def map(self, func):
        return Maybe(func(self.value)) if self.value is not None
else Maybe(None)

    def flat_map(self, func):
        return func(self.value) if self.value is not None else
Maybe(None)

    def __repr__(self):
        return f"Maybe({self.value})"

# Монада дозволяє зв'язувати виклики без вкладених перевірок
result = Maybe(5).flat_map(lambda x: Maybe(x * 2)).flat_map(lambda
x: Maybe(x + 1))
print(result)  # Maybe(11)
```

Функтори IO (I/O Functors)

Використовуються для роботи з зовнішнім світом (файли, API, введення/виведення). В чистому функціональному програмуванні (Haskell, Scala) вони використовуються, щоб ізолювати `I/O`. Функтори IO дозволяють ізолювати ефекти і робити код чистим.

Приклад `IO` функтора (емуляція в Python):

```
class IO:
    def __init__(self, effect):
        self.effect = effect

    def map(self, func):
```

```
        return IO(lambda: func(self.effect()))

    def run(self):
        return self.effect()

# Створення IO-ефекту (читаємо з файлу)
read_file = IO(lambda: open("test.txt").read())

# Використання map для перетворення рядка
uppercase_content = read_file.map(str.upper)

# Виконуємо ефект (читаємо файл та перетворюємо)
print(uppercase_content.run())
```

Порівняння категорій функторів

Тип функтора	Що робить?	Приклад
Коваріантний	Перетворює значення без зміни контейнера (`map()`).	`Box`, `List`
Контраваріантний	Перетворює вхідну функцію (`contramap()`).	`Printer`
Аплікативний	Застосовує функції, що знаходяться у контейнері (`apply()`).	`Maybe`, `Option`
Монада	Послідовно виконує обчислення (`flatMap()`).	`Maybe`, `IO`
IO-функтор	Використовується для обробки побічних ефектів (`run()`).	`IO`