**Лабораторна робота №1.**

**Основи функціонального програмування у Python**

**Ціль:**

* зрозуміти, що таке **чиста функція** і **референтна прозорість**;
* навчитися виявляти та ізолювати **побічні ефекти**;
* переписати імперативний фрагмент у функціональному стилі;
* застосувати typing.Callable для параметризації обчислень.

**Теорія**

* **Чиста функція:** детермінована (однаковий вхід → однаковий вихід), без змін глобального стану/вхідних даних, без I/O усередині.
* **Референтна прозорість:** вираз можна замінити його значенням без зміни поведінки програми.
* **Побічні ефекти:** I/O (print, файл, мережа), мутація вхідних структур, використання/зміна глобалів, залежність від time, random, os.environ тощо.
* **Підхід:** відокремити **чисте ядро** (перетворення даних) від **оболонки ефектів** (читання/запис).

**Стартовий імперативний фрагмент (який треба переписати)**

# starter\_imperative.py

TAX\_RATE = 0.2 # глобальний стан

def process\_orders(orders, min\_total, discount):

# orders: список словників {'id': int, 'items': [{'price': float, 'qty': int}], 'paid': bool}

valid = []

total\_revenue = 0.0

for o in orders:

if not o["paid"]:

continue

# побічний ефект

print("Processing order:", o["id"])

# мутація

order\_total = 0.0

for it in o["items"]:

order\_total += it["price"] \* it["qty"]

if order\_total < min\_total:

continue

# ще мутації і глобал TAX\_RATE

order\_total = order\_total \* (1 - discount)

order\_total = order\_total \* (1 + TAX\_RATE)

o["total"] = order\_total # мутуємо вхідні дані

valid.append(o)

total\_revenue += order\_total

return {"count": len(valid), "revenue": total\_revenue, "orders": valid}

**Проблеми:** глобал TAX\_RATE, print усередині, мутації o, змішане обчислення та I/O.

**Завдання 1. Переписати у функціональному стилі**

Перепишіть код так, щоб:

1. **Жодних мутацій** вхідних даних (створюйте нові об’єкти).
2. **Жодного I/O** всередині чистих функцій (ніяких print).
3. **Жодних глобальних констант** усередині обчислень - усі параметри передаються явно.

Підказка щодо структури:

# core.py — чисте ядро

from typing import Iterable, Callable, TypedDict, List, Dict

class Item(TypedDict):

price: float

qty: int

class Order(TypedDict, total=False):

id: int

items: List[Item]

paid: bool

total: float # з'явиться в результаті

def order\_subtotal(order: Order) -> float:

return sum(it["price"] \* it["qty"] for it in order["items"])

def with\_total(order: Order, total: float) -> Order:

# повертаємо НОВИЙ словник (вхідний не чіпаємо)

return {\*\*order, "total": total}

def process\_orders\_pure(

orders: Iterable[Order],

\*,

min\_total: float,

discount: float,

tax\_rate: float

) -> Dict[str, object]:

paid = (o for o in orders if o["paid"])

qualified = []

revenue = 0.0

for o in paid:

subtotal = order\_subtotal(o)

if subtotal < min\_total:

continue

total = subtotal \* (1 - discount)

total = total \* (1 + tax\_rate)

new\_o = with\_total(o, total)

qualified.append(new\_o)

revenue += total

return {"count": len(qualified), "revenue": revenue, "orders": qualified}

# app.py — оболонка побічних ефектів

from core import process\_orders\_pure

def run(orders):

result = process\_orders\_pure(orders, min\_total=100, discount=0.1, tax\_rate=0.2)

for o in result["orders"]:

print("Processed:", o["id"], "total:", o["total"])

print("Revenue:", result["revenue"])

**Завдання 2. Використати typing.Callable**

Зробіть обчислення **параметризованим** за допомогою функцій, що передаються як аргументи. Наприклад, винесіть політику знижок і податків у функції:

from typing import Callable

SubtotalFn = Callable[[float], float] # тож можна вставити купони

DiscountFn = Callable[[float], float] # (subtotal) -> discounted

TaxFn = Callable[[float], float] # (discounted) -> with tax

FilterFn = Callable[[float], bool] # (subtotal) -> bool

def make\_processor(

\*,

accept: FilterFn,

apply\_discount: DiscountFn,

apply\_tax: TaxFn,

) -> Callable[[list[Order]], dict[str, object]]:

def process(orders: list[Order]) -> dict[str, object]:

qualified = []

revenue = 0.0

for o in orders:

if not o["paid"]:

continue

subtotal = order\_subtotal(o)

if not accept(subtotal):

continue

total = apply\_tax(apply\_discount(subtotal))

qualified.append(with\_total(o, total))

revenue += total

return {"count": len(qualified), "revenue": revenue, "orders": qualified}

return process

# Приклад конфігурації:

accept = lambda s: s >= 100

apply\_discount = lambda s: s \* (1 - 0.1)

apply\_tax = lambda s: s \* (1 + 0.2)

processor = make\_processor(accept=accept, apply\_discount=apply\_discount, apply\_tax=apply\_tax)

Так ми отримуємо **компонувальний** (composable) підхід: будь-яку політику легко замінити, передавши інші функції.

**Завдання 3. Виявлення та ізоляція побічних ефектів**

1. Перенесіть print/логування з циклу в оболонку (наприклад, функцію render\_report(result)), яка **тільки споживає** чистий результат.
2. Якщо потрібно random або time, передавайте генератор/постачальник через Callable (інжекція залежностей), щоб ядро лишалося чистим:

from typing import Callable

NowFn = Callable[[], float]

def stamp\_total(total: float, now: NowFn) -> tuple[float, float]:

# чиста щодо total, а "час" прийшов ззовні

return total, now()

**Додаткові вправи**

* Напишіть compose і побудуйте знижку з кількох маленьких функцій:

from typing import Callable, TypeVar

A = TypeVar("A"); B = TypeVar("B"); C = TypeVar("C")

def compose(f: Callable[[B], C], g: Callable[[A], B]) -> Callable[[A], C]:

return lambda x: f(g(x))

* Створіть фабрику множників: def make\_multiplier(k: int) -> Callable[[int], int]: ...
* Перепишіть будь-яку функцію з мутаціями списку на повернення **нового** списку (list comprehension / map+filter - без побічних ефектів).

**Тести (мінімум)**

pytest-тести для чистого ядра:

def test\_referential\_transparency(sample\_orders):

from core import process\_orders\_pure

args = dict(min\_total=100, discount=0.1, tax\_rate=0.2)

r1 = process\_orders\_pure(sample\_orders, \*\*args)

r2 = process\_orders\_pure(sample\_orders, \*\*args)

assert r1 == r2 # той самий вхід → той самий вихід

def test\_no\_mutation(sample\_orders):

from copy import deepcopy

from core import process\_orders\_pure

original = deepcopy(sample\_orders)

process\_orders\_pure(sample\_orders, min\_total=0, discount=0.0, tax\_rate=0.0)

assert sample\_orders == original # вхід не змінюється

def test\_callable\_policies(sample\_orders):

from core import order\_subtotal, with\_total

from typing import Callable

DiscountFn = Callable[[float], float]

TaxFn = Callable[[float], float]

accept = lambda s: s >= 50

apply\_discount: DiscountFn = lambda s: s \* 0.9

apply\_tax: TaxFn = lambda s: s \* 1.2

# зберіть make\_processor і перевірте, що revenue рахується коректно

**Формат здачі**

lab01/

README.md # що зроблено, як запускати

core.py # ЧИСТІ функції

app.py # I/Oоболонка

tests/test\_core.py

requirements.txt (pytest, mypy, black, ruff)

**Критерії оцінювання (10 балів)**

* **Коректність** (4): обчислення правильні, тести проходять.
* **Функц. стиль / чистота** (3): відсутність мутацій/глобалів/I-O у ядрі, параметризація через Callable.
* **Тестування та типи** (2): є pytest-тести, анотації типів, mypy без помилок.
* **Читабельність** (1): зрозуміла структура, імена, black/ruff чисто.