ЗАТВЕРДЖЕНО

Науково-методичною радою
Державного університету
«Житомирська політехніка»

протокол від 19 травня 2026 р.
№ 4

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ
для проведення лабораторних занять з вибіркової навчальної дисципліни фахової підготовки «Програмування мовою Python»
Частина 1. Основи Python

Рекомендовано на засіданні
кафедри КТуМтаТ
8 квітня 2026 р., протокол № 3

Розробники: ст. викл. кафедри КТуМтаТ МОРОЗОВ Дмитро
к.т.н., доцент кафедри КТуМтаТ КОЛОМІЄЦЬ Роман
ст. викл. кафедри КітаКБ ОКУНЬКОВА Оксана
ст. викл. кафедри КН ЖЕЛІЗКО Віктор

Житомир
2026

ЗМІСТ

ВСТУП

Мова програмування Python сьогодні є однією з найпопулярніших і найзатребуваніших у світі. Її обирають і досвідчені розробники великих технологічних компаній, і дослідники університетів, і школярі, що лише починають знайомство з програмуванням. Причина такої універсальності – у вдалому поєднанні: синтаксис Python читається майже як звичайна мова, а потужність інструментів, що надає його екосистема, не поступається жодній іншій мові. Навчитися Python – означає отримати інструмент, придатний для вирішення задач у веб-розробці, аналізі даних, автоматизації, науці та інженерії.

Перша частина цього збірника присвячена **фундаменту мови** – тим знанням і навичкам, без яких неможливо рухатися далі. Вісім лабораторних робіт охоплюють увесь спектр базових концепцій. **Лабораторна робота 1** знайомить із основами: змінними, типами даних, введенням і виведенням, математичними операціями та керуючими конструкціями – умовними операторами і циклами. **Лабораторна робота 2** поглиблює розуміння послідовностей: рядків та списків – двох найуживаніших типів у повсякденному коді. **Лабораторна робота 3** розкриває словники, множини та кортежі – структури, що дозволяють організовувати складніші набори даних.

Лабораторна робота 4 присвячена функціям: аргументам, замиканням, декораторам і лямбда-виразам – інструментам, що роблять код повторно використовуваним і виразним. **Лабораторна робота 5** навчає правильно поводитися з помилками: конструкції try-except, кастомні виняткові ситуації та логування подій – навички, які відрізняють надійний код від крихкого. **Лабораторна робота 6** охоплює роботу з файловою системою: читання і запис файлів, кодування тексту, безпечне переміщення і копіювання, пошук через регулярні вирази. **Лабораторні роботи 7 і 8** завершують першу частину, вводячи об'єктно-орієнтоване програмування та автоматичне тестування – два підходи, що є основою сучасної розробки програмного забезпечення.

Кожна лабораторна робота побудована за єдиним принципом: спочатку теоретичні відомості з поясненнями і прикладами коду, потім практичні завдання різних рівнів складності – від простих до комплексних, – і нарешті контрольні запитання для самоперевірки. Такий підхід дає змогу як повним початківцям, так і студентам з певним досвідом знаходити для себе відповідний рівень виклику і рухатися вперед у власному темпі. Завдання третього рівня складності свідомо виходять за межі простого відтворення матеріалу – вони вимагають самостійного мислення і творчого застосування знань.

Опанування першої частини – це не просто знайомство з синтаксисом конкретної мови. Це формування **алгоритмічного мислення**: вміння розбивати задачу на кроки, обирати правильну структуру даних, передбачати помилки і писати код, який легко читати і підтримувати. Ці навички залишаться з вами незалежно від того, яку мову програмування ви використовуватимете у майбутньому. Python є чудовим середовищем для їх набуття: мова прозора, поведінка передбачувана, а зворотний зв'язок – миттєвий. Кожна успішно виконана програма – це маленька перемога, що додає впевненості і мотивації рухатися далі.

Вимоги до виконання та оформлення звітів з лабораторних робіт

Підготовка до кожної роботи проводиться в позааудиторний час.

Студенти опрацьовують лекційний матеріал, знайомляться із загальними відомостями, пишуть необхідні програми відповідно до отриманого варіанта індивідуального завдання.

Усі роботи виконуються на мові програмування Python. Під час занять студенти проводять тестування написаних програм, тобто запускають їх в інтегрованому середовищі розробки на персональних комп'ютерах, займаються налагодженням і виконують необхідні розрахунки.

Якщо студент виконує завдання лабораторних робіт на власному комп'ютері, він має самостійно встановити інтерпретатор мови Python (www.python.org/downloads) та редактор або інтегроване середовище розробки. В якості середовища розробки студенту пропонується використовувати PyCharm (www.jetbrains.com/pycharm) або редактор Visual Studio Code (code.visualstudio.com) чи будь які інші редактори коду для роботи з мовою Python з відповідними програмними модулями і налаштуваннями.

Після виконання роботи студент оформляє звіт, який складається з таких частин:

1. Титульний аркуш
2. Тема і мета роботи
2. Завдання на лабораторну роботу
3. Текст програми
4. Результати виконання програми (скріншоти терміналу, графіки тощо)
5. Висновки.

Приклад оформлення звіту про виконання лабораторної роботи наведено на сторінці дисципліни на освітньому порталі.

Титульний аркуш має містити інформацію про назву дисципліни, номер і тему лабораторної роботи, прізвище, ім'я і по-батькові виконавця, шифр і номер його групи і шифр роботи в рамці основного напису.

В шифрі роботи потрібно зазначити:

1. Шифр групи (виділено синьою рамкою).
2. Номер варіанту індивідуального завдання, якщо воно передбачено в роботі (виділено зеленою рамкою).
3. Три останні цифри залікової книжки (виділено червоною рамкою)

Робота оформлюється в електронному вигляді і надсилається в форматі pdf разом з файлами з кодом програм відповідних завдань лабораторної роботи на адресу електронної пошти викладача для перевірки.

Для кожного завдання пишеться окрема програма на Python. Імена файлів з кодом мають відповідати номеру завдання: *task1.1.py*, *task1.2.py* тощо.

Під час захисту роботи необхідно відповісти на контрольні питання і вміти пояснити роботу програми.

Лабораторна робота №1.

Основи Python: змінні, типи даних та керуючі конструкції

Мета роботи: Ознайомитися з основними концепціями мови Python. Вивчити конструкції розгалуження; зрозуміти роль логічних виразів та операторів порівняння як основи умовної логіки. Опанувати цикли `while` і `for`, функцію `range()`, оператори `break` і `continue`, а також навчитися застосовувати отримані знання для написання програм, що поєднують введення даних, обчислення та розгалужену логіку.

Теоретичні відомості:

Частина І. Змінні, типи даних та введення-виведення

Функція `print()` виводить інформацію на екран. Нижче будуть приводитись приклади коду на мові Python з поясненнями.

```
# Виведення рядка тексту
print('Привіт!')

# Виведення кількох значень через кому (роздільник – пробіл)
print('Сума:', 3 + 5)        # Сума: 8

# Параметр sep – роздільник між значеннями
print('a', 'b', 'c', sep='-')   # a-b-c

# Параметр end – символ кінця рядка (за замовч. '\n')
print('Рядок 1', end=' ')
print('Рядок 2')              # Рядок 1 Рядок 2
```

Змінна – це іменована область пам'яті для зберігання даних. У Python не потрібно оголошувати тип – він визначається автоматично при присвоєнні.
Правила іменування: може містити літери, цифри та _, не може починатися з цифри, регістр важливий (`age` та `Age` – різні змінні).

```
# Присвоєння значень
name = 'Олена'
age = 20
height = 1.68
is_student = True

# Множинне присвоєння
x = y = z = 0

# Паралельне присвоєння
a, b, c = 1, 2, 3

# Swap (обмін значень) – без тимчасової змінної
a, b = b, a
print(a, b)  # 2 1

print(name, age, height)
```

Типи даних та перетворення

Тип	Опис та приклад
`int`	Ціле число: `age = 21, temp = -5`
`float`	Дробове число: `price = 29.99, pi = 3.14`
`str`	Рядок тексту: `name = 'Іван'`
`bool`	Логічний тип: `True` або `False`
`None`	Відсутність значення: `result = None`

```
# Перевірка типу через type()
x = 42
print(type(x))              # <class 'int'>
print(type(3.14))           # <class 'float'>
print(type('hello'))        # <class 'str'>
print(type(True))           # <class 'bool'>

# Явне перетворення типів
n = int('42')               # рядок -> int
f = float('3.14')           # рядок -> float
s = str(100)                # int    -> str
b = bool(0)                 # int    -> bool  (False)

# Небезпечне перетворення – звертаємо увагу!
print(int(3.9))             # 3  (не 4! відкидає дробову частину)
print(bool(''))             # False  (порожній рядок – False)
print(bool('0'))            # True   (непорожній рядок – True!)
```

Функція `input()` зупиняє виконання програми та чекає введення з клавіатури. Вона ЗАВЖДИ повертає рядок (str) – навіть якщо введено число.

```
Базове введення
name = input('Введіть ваше ім\'я: ')
print('Привіт,', name)

# ВАЖЛИВО: результат input() – завжди рядок!
age_str = input('Введіть вік: ')
age = int(age_str)  # обов'язково конвертуємо

# Або одразу:
age = int(input('Вік: '))
height = float(input('Зріст (м): '))

# Повна програма: введення і виведення
name = input('Ім\'я: ')
year = int(input('Рік народження: '))
age = 2025 - year
print(f'Привіт, {name}! Тобі {age} років.')
```

Математичні операції та модуль `math`

```python
a, b = 15, 4

print(a + b)    # 19   - додавання
print(a - b)    # 11   - віднімання
print(a * b)    # 60   - множення
print(a / b)    # 3.75 - ділення (завжди float)
print(a // b)   # 3    - ціле ділення
print(a % b)    # 3    - остача від ділення
print(a ** b)   # 50625 - степінь

# Скорочені оператори присвоєння
x = 10
x += 5    # x = 15
x -= 3    # x = 12
x *= 2    # x = 24
x //= 7   # x = 3

# Модуль math
import math

print(math.pi)            # 3.14159...
print(math.sqrt(49))      # 7.0
print(math.floor(4.7))    # 4  - вниз
print(math.ceil(4.2))     # 5  - вгору
print(math.pow(2, 10))    # 1024.0
print(math.log(100, 10))  # 2.0
print(math.sin(math.pi/2))  # 1.0
print(math.radians(180))  # 3.14159...
```

F-рядки та форматування

```python
name = 'Іван'
score = 94.333

# f-рядок: вираз у фігурних дужках
print(f'Студент: {name}')
print(f'Бал: {score:.2f}')  # 2 знаки після коми
print(f'Сума: {2 + 3}')     # обчислення у {}

# Вирівнювання
for item, price in [('Молоко', 38.5), ('Хліб', 22), ('Масло', 115)]:
    print(f'{item:<10} {price:>7.2f} грн')
# Молоко       38.50 грн
# Хліб         22.00 грн
# Масло       115.00 грн

# Коментарі у коді
# Це однорядковий коментар
x = 5  # коментар після коду
```

Частина 2. Керуючі конструкції

Основою будь-якої умови є логічний вираз – вираз, що повертає `True` або `False`.

```
a, b = 10, 20

# Оператори порівняння
print(a == b)    # False – рівність
print(a != b)    # True  – нерівність
print(a < b)     # True  – менше
print(a > b)     # False – більше
print(a <= 10)   # True  – менше або рівне

# Логічні оператори
x = 15
print(x > 10 and x < 20)   # True  – обидві умови True
print(x < 5  or  x > 10)   # True  – хоча б одна True
print(not (x > 10))        # False – інверсія

# Оператор in – перевірка входження
month = 7
print(month in [6, 7, 8])   # True – місяць є у списку
```

Умовний оператор `if-elif-else`

Відступи (4 пробіли) є синтаксичною вимогою Python – вони визначають, які рядки належать до блоку.

```
# if-else
age = int(input('Введіть вік: '))
if age >= 18:
    print('Доступ дозволено')
else:
    print('Доступ заборонено')

# if-elif-else: кілька варіантів
score = int(input('Оцінка (0-100): '))

if score >= 90:
    grade = 'Відмінно'
elif score >= 75:
    grade = 'Добре'
elif score >= 60:
    grade = 'Задовільно'
else:
    grade = 'Незадовільно'

print(f'Оцінка: {grade}')

# Перевірка типу трикутника (вкладені if)
a = float(input('Сторона a: '))
b = float(input('Сторона b: '))
c = float(input('Сторона c: '))

if a + b > c and a + c > b and b + c > a:
    if a == b == c:
        print('Рівносторонній')
    elif a == b or b == c or a == c:
        print('Рівнобедрений')
```

```
    else:
        print('Різносторонній')
else:
    print('Трикутник не існує')
```

Тернарний оператор – скорочена форма `if-else` для написання в одному рядку.

```
# Тернарний оператор
x = 7
result = 'парне' if x % 2 == 0 else 'непарне'
print(result)              # непарне

# Максимум двох чисел
a, b = 15, 28
maximum = a if a > b else b
print(f'Максимум: {maximum}')  # 28

# match-case (Python 3.10+) – аналог switch
day = int(input('Введіть номер дня (1-7): '))

match day:
    case 1:
        print('Понеділок')
    case 2:
        print('Вівторок')
    case 3:
        print('Середа')
    case 4:
        print('Четвер')
    case 5:
        print('П\'ятниця')
    case 6 | 7:            # кілька значень через |
        print('Вихідний')
    case _:                # за замовчуванням
        print('Некоректний день')
```

Цикл `while` виконує тіло поки умова `True`.

```
# Підрахунок від 1 до 5
i = 1
while i <= 5:
    print(i, end=' ')   # 1 2 3 4 5
    i += 1              # змінюємо лічильник

# Введення до правильного значення
while True:
    age = int(input('Вік (1-120): '))
    if 1 <= age <= 120:
        break           # вихід при правильному введенні
    print('Некоректне значення')
print(f'Ваш вік: {age}')

# Сума цифр числа
n = int(input('Число: '))
total = 0
while n > 0:
    total += n % 10     # остання цифра
    n //= 10            # відкидаємо останню цифру
print(f'Сума цифр: {total}')
```

Цикл `for` перебирає елементи послідовності. Функція `range()` генерує числовий діапазон.

```
# range(stop) – від 0 до stop-1
for i in range(5):
    print(i, end=' ')      # 0 1 2 3 4

# range(start, stop)
for i in range(1, 6):
    print(i, end=' ')      # 1 2 3 4 5

# range(start, stop, step) – з кроком
for i in range(0, 20, 5):
    print(i, end=' ')      # 0 5 10 15

# Зворотній відлік
for i in range(5, 0, -1):
    print(i, end=' ')      # 5 4 3 2 1

# Перебір рядка (рядок – послідовність символів)
for char in 'Python':
    print(char, end='-')   # P-y-t-h-o-n-

# Вкладені цикли: таблиця множення
for i in range(1, 4):
    for j in range(1, 4):
        print(f'{i}*{j}={i*j:2}', end='   ')
    print()    # переносимо рядок
# 1*1= 1   1*2= 2   1*3= 3
# 2*1= 2   2*2= 4   2*3= 6
# 3*1= 3   3*2= 6   3*3= 9
```

Оператори `break`, `continue` та `else` у циклах

```
# break – негайний вихід із циклу
numbers = [5, 12, 3, -7, 8, -2]
for num in numbers:
    if num < 0:
        print(f'Перше від\'ємне: {num}')
        break
    print(f'{num} – позитивне')

# continue – пропуск поточної ітерації
for i in range(1, 11):
    if i % 2 != 0:
        continue        # пропускаємо непарні
    print(i, end=' ')   # 2 4 6 8 10

# else у циклі – виконується якщо не було break
target = 7
numbers = [4, 2, 9, 1, 6]
for num in numbers:
    if num == target:
        print(f'Знайдено: {target}')
        break
else:
    print(f'{target} не знайдено')
```

Зверніть увагу!

`for x in range(n):` – виконується рівно n разів.

`while True:` з `break` – використовується для валідації введення.
`continue` пропускає залишок тіла циклу, але НЕ завершує цикл.
`break` у вкладеному циклі виходить тільки з ВНУТРІШНЬОГО циклу.

Завдання для лабораторної роботи

Для кожного завдання напишіть окрему програму на Python. Зберігайте файли з іменами task1.py … task7.py.

1.1. Оголосіть по дві змінні кожного з типів: `int, float, str, bool`. Виведіть значення та тип кожної через `print()` та `type()`. Виконайте перетворення типів: рядок у число, число у рядок. Продемонструйте параметри `print()`: `sep` та `end` на кількох прикладах.

1.2. Напишіть програму, яка зчитує два числа (`float`) і виводить результати всіх 7 арифметичних операцій (+, -, *, /, //, %, **). Дробові результати форматуйте до 2 знаків після коми через f-рядки. Також обчисліть і виведіть гіпотенузу прямокутного трикутника за двома катетами (`math.sqrt`).

1.3. Напишіть дві програми:
- зчитує ціле число і визначає через `if-elif-else`: чи воно додатне/від'ємне/нульове; парне або непарне; чи ділиться одночасно на 3 і на 5. Використайте тернарний оператор хоча б один раз;
- зчитує номер місяця (1–12) і через `match-case` виводить пору року та кількість днів у місяці (для лютого – 28).

1.4. Напишіть програму, яка зчитує числа від користувача по одному. Введення числа 0 зупиняє цикл (використовуйте `while True` з `break`). Після завершення виведіть: суму, кількість введених чисел, середнє арифметичне, мінімум та максимум, кількість від'ємних та додатних чисел.

1.5. Реалізуйте через `for` та `range()`:
- виведіть всі прості числа від 2 до N (N вводить користувач). Для перевірки простоти використайте вкладений цикл із break та else;
- побудуйте трикутник Паскаля для перших 6 рядків. Кожен елемент – сума двох елементів над ним. Виведіть у форматованому вигляді.

1.6. Напишіть програму-калькулятор у вигляді циклу:
- програма запитує два числа та символ операції (+, -, *, /, //, %, **);
- обчислює та виводить результат через match-case;
- захист від ділення на нуль для / та //;
- після кожного обчислення питає: 'Продовжити? (y/n)' – при 'n' завершує роботу;
- рахує кількість виконаних обчислень і виводить її при завершенні.

1.7. Реалізуйте програму в двох частинах:
Частина 1 – Гра «Вгадай число»:
- задайте «секретне» число (наприклад, 42) у коді;
- цикл `while` запитує у користувача число від 1 до 100;

- після кожної спроби виводити підказку «більше» або «менше»;
- обмежте кількість спроб до 7; при вичерпанні – вивести відповідь через `else` у `while`;
- при вгадуванні – привітання і кількість спроб.

Частина 2 – Статистика числової послідовності:

- зчитати N чисел (N вводить користувач) через `for i in range(N)`;
- обчислити та вивести: суму, середнє, мінімум, максимум (без вбудованих `min/max/sum` – вручну через цикл);
- підрахувати кількість чисел кратних 3, кратних 5 і кратних одночасно 3 та 5;
- вивести числа у зворотньому порядку (зберегти у список через append і вивести у зворотному порядку).

Контрольні запитання

1. Що таке змінна? Які є правила іменування змінних у Python? Чи важливий регістр?
2. Назвіть основні вбудовані типи даних Python. Чим int відрізняється від `float`? Що таке `bool`?
3. Що завжди повертає функція `input()`? Чому `print(input('Число: ') + 1)` спричиняє помилку і як її виправити?
4. Яка різниця між оператором ділення `/` та цілочисельного ділення `//`? Що поверне вираз `17 % 5`?
5. Що таке правило LEGB у Python? (Local, Enclosing, Global, Built-in) – поясніть на прикладі змінної з різними значеннями.
6. Чим відрізняється `if-elif-else` від кількох окремих `if`? Коли доцільно використовувати `match-case`?
7. Що таке тернарний оператор? Запишіть вираз для знаходження максимуму двох чисел `a` і `b`.
8. Чим відрізняється цикл `while` від `for`? Наведіть задачі де кожен є кращим вибором.
9. Що роблять оператори `break` і `continue`? Коли виконується блок `else` у циклі?
10. Напишіть програму (3–5 рядків), яка зчитує радіус кола, обчислює площу та периметр і виводить результати.

Лабораторна робота №2.

Рядки та списки

Мета роботи: Вивчити рядковий тип `str`: способи створення, індексування, зрізи, базові операції та рядкові методи для пошуку, заміни, зміни регістру, розбиття та об'єднання рядків. Ознайомитися з типом `list`: створення списків, індексування, зрізи, методи додавання і видалення елементів, сортування та пошук. Навчитися застосовувати спільні для рядків і списків механізми (зрізи, in, len, for, enumerate) та розуміти, коли використовувати кожен з типів для розв'язання практичних задач.

Теоретичні відомості:

Рядки (`str`) та списки (`list`) є двома найуживанішими типами послідовностей у Python. Вони поділяють багато спільних операцій, але мають принципову відмінність: рядок незмінний, список – змінюваний.

Властивість	**str (рядок)**	**list (список)**
Тип елементів	Символи Unicode	Будь-які об'єкти
Змінюваність	Незмінний (immutable)	Змінюваний (mutable)
Індексування	`s[0], s[-1]`	`lst[0], lst[-1]`
Зрізи	`s[1:4]`	`lst[1:4]`
Конкатенація	`s1 + s2`	`lst1 + lst2`
Повторення	`s * 3`	`lst * 3`
Входження	`'py' in s`	`3 in lst`
Довжина	`len(s)`	`len(lst)`
Ітерація	`for c in s:`	`for x in lst:`

Створення рядків. Спецсимволи та f-рядки

```
# Одинарні, подвійні та потрійні лапки
s1 = 'Привіт'
s2 = "Привіт"
s3 = '''Перший рядок
Другий рядок'''

# Escape-послідовності
s4 = 'Рядок 1\nРядок 2'    # \n – новий рядок
s5 = 'A\tB\tC'             # \t – табуляція
s6 = 'Вона сказала: \'OK\'' # \' – лапка всередині
s7 = r'C:\Users\file.txt'  # raw-рядок: \ не є escape

# f-рядки: вирази у фігурних дужках
name, gpa = 'Іван', 4.2
print(f'Студент: {name}, GPA: {gpa:.2f}')

# Порожній рядок
empty = ''
print(len(empty))  # 0
```

Кожен символ має числовий індекс. Додатні– з початку (від 0), від'ємні – з кінця (від -1).

```
word = 'Python'
#       P  y  t  h  o  n
# +idx  0  1  2  3  4  5
# -idx -6 -5 -4 -3 -2 -1

print(word[0])       # P
print(word[-1])      # n  - останній символ
print(word[2])       # t

# Зріз: s[start : stop : step]
s = 'Hello, World!'
print(s[0:5])        # Hello
print(s[7:])         # World!
print(s[:5])         # Hello
print(s[-6:])        # World!
print(s[::2])        # Hlo ol!  - кожен другий
print(s[::-1])       # !dlroW ,olleH  - реверс

# Рядок незмінний – це спричинить TypeError:
# word[0] = 'p'   # TypeError: 'str' does not support item assignment
```

Рядкові методи: регістр та пошук

Метод / Синтаксис	Опис / Приклад
`s.upper()`	`'hello'.upper()` → `'HELLO'`
`s.lower()`	`'WORLD'.lower()` → `'world'`
`s.title()`	`'py code'.title()` → `'Py Code'`
`s.capitalize()`	`'hELLO'.capitalize()` → `'Hello'`
`s.swapcase()`	`'PyThOn'.swapcase()` → `'pYtHoN'`
`s.find(sub)`	`'python'.find('th')` → `2`; `-1` якщо немає
`s.index(sub)`	Як `find()`, але кидає `ValueError` якщо немає
`s.count(sub)`	`'banana'.count('a')` → `3`
`s.startswith(p)`	`'file.txt'.startswith('file')` → `True`
`s.endswith(p)`	`'file.txt'.endswith('.txt')` → `True`
`s.isdigit()`	`'123'.isdigit()` → `True`
`s.isalpha()`	`'abc'.isalpha()` → `True`
`s.isspace()`	`'  '.isspace()` → `True`

```
email = 'user@example.com'

# Пошук позиції @ та виділення частин
pos = email.find('@')
print(email[:pos])      # user
print(email[pos+1:])    # example.com

# Безпечна перевірка перед index()
target = 'xyz'
if target in email:
    print(email.index(target))
else:
    print('Не знайдено')

# Перевірка рядка-числа (для безпечного int())
s = input('Введіть число: ')
```

```python
if s.lstrip('-').isdigit():
    print(f'Ціле число: {int(s)}')
else:
    print('Це не ціле число')
```

Рядкові методи: очищення, заміна, розбиття

Метод / Синтаксис	Опис / Приклад
`s.strip()`	`'  hi  '.strip()` → `'hi'`
`s.lstrip() / rstrip()`	Очищення лише з лівого / правого краю
`s.replace(old, new)`	`'кіт сів'.replace('кіт','пес')` → `'пес сів'`
`s.replace(o,n,n)`	Замінити не більше n входжень
`s.split(sep)`	`'a,b,c'.split(',')` → `['a','b','c']`
`s.split()`	Розбиття за пробілами: ігнорує багаторазові
`sep.join(lst)`	`', '.join(['a','b'])` → `'a, b'`
`s.splitlines()`	Розбиття за символами нового рядка

```python
# Очищення введення
raw  = '  Іван Петренко  '
name = raw.strip()
print(f'|{name}|')          # |Іван Петренко|

# Розбиття і збирання
sentence = 'раз два три чотири'
words    = sentence.split()           # ['раз','два','три','чотири']
reversed_text = ' '.join(words[::-1]) # 'чотири три два раз'
print(reversed_text)

# CSV-розбір
csv_line = 'Іван,22,Київ,відмінник'
parts    = csv_line.split(',')
print(parts[0], parts[2])   # Іван Київ

# Підрахунок голосних
word   = 'програмування'
vowels = 'аеєиіїоуюя'
count  = sum(1 for ch in word if ch.lower() in vowels)
print(f'Голосних: {count}')
```

Перебір рядка. `ord()` та `chr()`

```python
# Перебір символів
for char in 'Python':
    print(char, end=' ')    # P y t h o n

# Перебір з індексом – enumerate()
for i, char in enumerate('Python', start=1):
    print(f'{i}:{char}', end=' ')  # 1:P 2:y 3:t 4:h 5:o 6:n

# ord() – Unicode-код символу
print(ord('A'))    # 65
print(ord('a'))    # 97

# chr() – символ за кодом
print(chr(65))     # A
print(chr(1055))   # П
```

```python
# Шифр Цезаря (зсув на 3)
text      = 'hello'
encrypted = ''.join(chr(ord(c) + 3) for c in text)
print(encrypted)   # khoor

# Вирівнювання рядків
s = 'Python'
print(s.center(14, '-'))   # ----Python----
print(s.ljust(12, '.'))    # Python......
print(s.rjust(12, '.'))    # ......Python
print(str(42).zfill(6))    # 000042
```

Список – впорядкована змінювана колекція елементів будь-яких типів. Головна відмінність від рядка: елементи можна додавати, видаляти та змінювати.

```python
# Способи створення
empty   = []
numbers = [3, 1, 4, 1, 5, 9, 2, 6]
mixed   = [42, 3.14, 'текст', True, None]
nested  = [[1, 2], [3, 4], [5, 6]]      # вкладені списки
chars   = list('Python')                 # ['P','y','t','h','o','n']
rng     = list(range(5))                 # [0, 1, 2, 3, 4]

# Атрибути
print(len(numbers))     # 8
print(type(numbers))    # <class 'list'>

# Конкатенація та повторення
a = [1, 2, 3]
b = [4, 5, 6]
print(a + b)            # [1, 2, 3, 4, 5, 6]
print([0] * 5)          # [0, 0, 0, 0, 0]

# Перевірка входження
print(3 in a)           # True
print(7 not in a)       # True

# Агрегуючі функції (для числових списків)
nums = [5, 2, 8, 1, 9]
print(sum(nums))        # 25
print(min(nums))        # 1
print(max(nums))        # 9
```

Індексування та зрізи списку

```python
lst = [10, 20, 30, 40, 50, 60]

# Звернення до елементів
print(lst[0])       # 10
print(lst[-1])      # 60  – останній

# Зрізи – аналогічно до рядків
print(lst[1:4])     # [20, 30, 40]
print(lst[:3])      # [10, 20, 30]
print(lst[::2])     # [10, 30, 50]  – кожен другий
```

```python
print(lst[::-1])    # [60, 50, 40, 30, 20, 10]  - реверс

# КЛЮЧОВА ВІДМІННІСТЬ від рядка: списки змінювані!
lst[2] = 99
print(lst)          # [10, 20, 99, 40, 50, 60]

# Заміна зрізу
lst[1:3] = [200, 300]
print(lst)          # [10, 200, 300, 40, 50, 60]

# Копія через зріз (НЕ псевдонім!)
original = [1, 2, 3]
copy     = original[:]
copy[0]  = 99
print(original)     # [1, 2, 3]  - незмінний
print(copy)         # [99, 2, 3]

# Псевдонім (обидві змінні - один об'єкт)
alias    = original
alias[0] = 99
print(original)     # [99, 2, 3]  - ЗМІНИВСЯ!
```

Методи списку: додавання та видалення

Метод / Синтаксис	Опис / Приклад
`lst.append(x)`	Додає `x` у кінець: `[1,2].append(3)` → `[1,2,3]`
`lst.insert(i, x)`	Вставляє `x` на позицію `i`
`lst.extend(iter)`	Розширює елементами ітерованого (аналог `+=`)
`lst.remove(x)`	Видаляє перше входження `x` (`ValueError` якщо немає)
`lst.pop()`	Видаляє `i` повертає останній елемент
`lst.pop(i)`	Видаляє `i` повертає елемент за індексом `i`
`del lst[i]`	Видаляє елемент (або зріз) за індексом
`lst.clear()`	Видаляє всі елементи: `lst` стає `[]`
`lst.index(x)`	Індекс першого входження `x`
`lst.count(x)`	Кількість входжень `x`
`lst.copy()`	Поверхнева копія (аналог `lst[:]`)
`lst.reverse()`	Реверсує список НА МІСЦІ (повертає None)

```python
lst = [1, 2, 3]

lst.append(4)           # [1, 2, 3, 4]
lst.insert(0, 0)        # [0, 1, 2, 3, 4]  - на початок
lst.extend([5, 6])      # [0, 1, 2, 3, 4, 5, 6]

lst.remove(0)           # [1, 2, 3, 4, 5, 6]
last  = lst.pop()       # last=6, lst=[1,2,3,4,5]
third = lst.pop(2)      # third=3, lst=[1,2,4,5]

del lst[1:3]            # lst=[1, 5]

# Безпечне видалення через перевірку
if 99 in lst:
    lst.remove(99)

# Знаходження всіх позицій значення
grades   = [85, 92, 78, 92, 88, 92]
all_pos  = [i for i, g in enumerate(grades) if g == 92]
print(all_pos)          # [1, 3, 5]
```

Сортування списків

```python
nums  = [5, 2, 8, 1, 9, 3]
words = ['банан', 'ківі', 'полуниця', 'яблуко']

# .sort() – сортує НА МІСЦІ, повертає None
nums.sort()
print(nums)                # [1, 2, 3, 5, 8, 9]
nums.sort(reverse=True)
print(nums)                # [9, 8, 5, 3, 2, 1]

# sorted() – повертає НОВИЙ список, оригінал незмінний
original = [3, 1, 4, 1, 5]
new_lst  = sorted(original)
print(original)            # [3, 1, 4, 1, 5]  – незмінний
print(new_lst)             # [1, 1, 3, 4, 5]

# Сортування за ключем (key=)
words.sort(key=len)        # за довжиною слова
print(words)               # ['ківі', 'банан', 'яблуко', 'полуниця']

# Сортування без урахування регістру
cities = ['Харків', 'київ', 'Одеса', 'львів']
cities.sort(key=str.lower)
print(cities)

# sorted() з lambda: за другим елементом пари
students = [('Марія', 95), ('Іван', 78), ('Олена', 88)]
by_score = sorted(students, key=lambda s: s[1], reverse=True)
print(by_score)  # [('Марія', 95), ('Олена', 88), ('Іван', 78)]
```

Списокове включення (list comprehension) – компактний синтаксис створення нового списку через вираз у квадратних дужках. Замінює цикл `for + append()`.

```python
# Базова форма: [вираз for змінна in послідовність]
squares = [x**2 for x in range(1, 6)]
print(squares)           # [1, 4, 9, 16, 25]

# З умовою: [вираз for змінна in послідовність if умова]
nums     = [1, -3, 4, -2, 7, -1, 5]
positive = [x for x in nums if x > 0]
print(positive)          # [1, 4, 7, 5]

# Перетворення рядків у список чисел
str_nums = ['1', '2', '3', '4', '5']
int_nums = [int(s) for s in str_nums]

# Сплощення вкладеного списку
nested = [[1, 2], [3, 4], [5, 6]]
flat   = [x for sub in nested for x in sub]
print(flat)              # [1, 2, 3, 4, 5, 6]

# enumerate() – перебір з індексом
fruits = ['яблуко', 'груша', 'банан']
for i, fruit in enumerate(fruits, start=1):
    print(f'{i}. {fruit}')

# zip() – перебір двох списків одночасно
names  = ['Іван', 'Марія', 'Олег']
```

```
scores = [88, 95, 72]
for name, score in zip(names, scores):
    print(f'{name}: {score}')
```

Рядки і списки тісно пов'язані: рядок можна перетворити на список символів, список рядків – об'єднати у рядок. Методи `split()` та `join()` є мостом між цими двома типами.

```
# Рядок → список символів
word = 'Python'
chars = list(word)          # ['P','y','t','h','o','n']

# Список символів → рядок
back = ''.join(chars)       # 'Python'

# split() і join() – найуживаніша пара
text  = 'яблуко груша банан вишня'
words = text.split()        # список слів
words.sort()                # сортуємо список
result = ' '.join(words)    # назад у рядок
print(result)               # 'банан вишня груша яблуко'

# Перевірка паліндрому через список
s = 'radar'
chars = list(s)
chars.reverse()
print(s == ''.join(chars))  # True

# Або через зріз (коротше):
print(s == s[::-1])         # True

# Рядок з форматованого списку
data = [('Іван', 88), ('Марія', 95), ('Олег', 72)]
lines = [f'{name}: {score}' for name, score in data]
report = '\n'.join(lines)
print(report)
# Іван: 88
# Марія: 95
# Олег: 72
```

Завдання на лабораторну роботу

2.1. Введіть будь-який рядок із клавіатури та виведіть:

- довжину, перший та останній символ;
- рядок у верхньому, нижньому регістрі та з великої літери кожного слова (`.title()`);
- рядок у зворотньому порядку (зріз);
- кількість входжень голосної літери 'а' (або іншої, яку введе користувач);
- чи починається і чи закінчується рядок на задану підстрічку;
- замінити всі пробіли на символ '_'.

2.2. Зчитайте 8 цілих чисел (кожне окремо через `input()`). Виведіть:

- весь список, його довжину, суму, мінімум та максимум;
- перший та останній елемент, перші три та останні три (зрізи);
- список у відсортованому порядку (`sorted()` – оригінал не змінювати!);

- список без дублікатів (підказка: `set()` перетворює на множину);
- елементи, що більше середнього арифметичного (спискове включення).

2.3. Зчитайте речення (рядок із кількох слів) та виконайте:
- розбийте на слова через `split()`;
- виведіть кількість слів, найдовше та найкоротше слово;
- відсортуйте слова за алфавітом і виведіть через кому та пробіл (`join`);
- виведіть слова у зворотньому порядку (реверс списку);
- знайдіть слова, що є паліндромами (`слово == слово[::-1]`).

2.4. Задайте список студентів у вигляді списку кортежів: (прізвище, середній_бал). Мінімум 8 студентів. Виконайте:
- відсортуйте за середнім бал) у спадному порядку (`sorted + lambda`);
- відсортуйте за прізвищем в алфавітному порядку;
- знайдіть студента з максимальним і мінімальним середнім балом (`min/max з key=lambda`);
- знайдіть всіх студентів з середній бал >= 4.0 (спискове включення);
- обчисліть середній бал групи (`sum + len` через спискове включення);
- сформуйте список лише прізвищ (спискове включення) і виведіть через join.

2.5. Реалізуйте задачі виключно через спискові включення (без циклів for з append):
- квадрати непарних чисел від 1 до 20;
- довжини слів із введеного речення, відфільтровані – тільки слова довжиною > 3;
- таблиця множення 5×5 як вкладений список: `[[i*j for j in range(1,6)] for i in range(1,6)]`;
- сплощення цієї матриці у один список;
- список пар (i, j) де i < j для i, j в діапазоні [1, 5].

2.6. Задайте текст (мінімум 5 речень) або зчитайте його по рядках (завершення – порожній рядок). Виконайте повний аналіз:
- підрахунок: символів (з пробілами і без), слів, речень, унікальних слів;
- частотний аналіз: топ-5 найуживаніших слів (без урахування регістру, очищених від розділових знаків через `strip('.,!?;:')`);
- знайти найдовше слово та слово, що зустрічається найчастіше;
- перевірити кожне слово: скільки є паліндромами;
- побудувати 'резюме': перші 80 символів тексту + '...' якщо текст довший;
- паралельно з текстовим аналізом: зчитайте список оцінок студентів у форматі `'Прізвище:оцінка'` (через `split(':')`). Виведіть: відсортований рейтинг, середню оцінку, кількість студентів вище та нижче середньої. Сформуйте фінальний рядок-звіт через `join()`.

Контрольні запитання

1. Чим рядок відрізняється від списку з точки зору змінюваності? Що станеться при спробі `s[0] = 'x'` для рядка?
2. Що таке зріз (slice)? Запишіть вираз для: 1) реверсу рядка s, 2) отримання кожного другого елемента, 3) останніх 3 символів.
3. Яка різниця між `.find()` та `.index()`? Коли кожен є кращим вибором?
4. Що повертає `s.split()` без аргументів? Чим відрізняється від `s.split(' ')`?
5. Поясніть різницю між `lst.sort()` та `sorted(lst)`. Наведіть приклад, коли `sorted()` є єдиним правильним вибором.
6. Що таке псевдонім (alias) списку? Чому `b = a` не створює копію і як правильно скопіювати список?
7. Що таке спискове включення (list comprehension)? Запишіть вираз, що повертає квадрати парних чисел від 2 до 20.
8. Для чого використовуються `enumerate()` та `zip()`? Наведіть приклад кожної функції.
9. Як пов'язані рядки і списки через методи `split()` та `join()`? Наведіть приклад сортування слів у реченні.
10. Які методи списку змінюють його на місці, а які повертають новий об'єкт? Наведіть 3 приклади кожного.

Лабораторна робота №3.

Словники, множини, кортежі

Мета роботи: Вивчити кортеж (tuple) як незмінну впорядковану послідовність; зрозуміти коли доцільно використовувати tuple замість list, навчитися застосовувати розпакування та використовувати кортежі як ключі словника.
Опанувати словник (dict) – найуживанішу структуру даних Python для зберігання пар ключ-значення: навчитися створювати словники, звертатися до елементів, додавати та видаляти пари, ітерувати та будувати генератори словників.
Вивчити множину (set та frozenset) для роботи з унікальними значеннями та операціями теорії множин (об'єднання, перетин, різниця); навчитися обирати правильну структуру даних для конкретної задачі.

Теоретичні відомості:

У попередній лабораторній роботі було розглянуто рядки (str) та списки (list). Python пропонує ще три важливих вбудованих типи – кортеж, словник та множину. Кожен з них має свої сильні сторони і призначений для певного кола задач.

Властивість	**dict**	**set**	**tuple**	**list (для порівняння)**
Змінюваний?	Так	Так	Ні	Так
Впорядкований?	Так (3.7+)	Ні	Так	Так
Дублікати?	Ключі: Ні	Ні	Так	Так
Синтаксис	{k: v}	{1,2,3}	(1,2)	[1,2]
Доступ	d[key]	in (O(1))	t[i]	l[i]
Типовий кейс	Ключ→значення	Унікальні	Незмінні дані	Змінна послідовність

Частина І. Кортежі (tuple)

Кортеж – це впорядкована незмінна послідовність. Він схожий на список, але після створення його елементи не можна змінити. Саме ця незмінність робить кортеж корисним: коли дані не мають мінятися – використання кортежу є явним сигналом про це іншим розробникам.

```
# Створення кортежів
empty   = ()                        # порожній
single  = (42,)                     # один елемент – КОМА обов'язкова!
wrong   = (42)                      # НЕ кортеж – це просто число 42
coords  = (10, 20)                  # дві координати
person  = ('Іван', 21, 4.2, True)   # різні типи – дозволено
no_paren = 1, 2, 3                  # дужки не обов'язкові

print(type(single))   # <class 'tuple'>
print(type(wrong))    # <class 'int'>  !

# Індексування та зрізи – як у списках
t = (10, 20, 30, 40, 50)
print(t[0])      # 10
print(t[-1])     # 50  – останній
print(t[1:4])    # (20, 30, 40)
print(t[::-1])   # (50, 40, 30, 20, 10)  – реверс
```

```python
# Спроба змінити – TypeError!
# t[0] = 99   # TypeError: 'tuple' object does not support item assignment

# Методи (їх лише два)
t = (1, 2, 2, 3, 2, 4)
print(t.count(2))    # 3  – кількість входжень
print(t.index(3))    # 3  – індекс першого входження

# len, in, перебір – як у списках
print(len(t))        # 6
print(2 in t)        # True
for x in t:
    print(x, end=' ')
print(int(3.9))
```

Розпакування (unpacking) – дуже зручна операція Python: присвоїти елементи кортежу кільком змінним за одну операцію. Ця техніка часто використовується при поверненні кількох значень з функцій.

```python
# Класичне розпакування
coords = (3, 7)
x, y = coords
print(x, y)        # 3 7

# Розпакування довільної кількості
first, *rest = (1, 2, 3, 4, 5)
print(first)       # 1
print(rest)        # [2, 3, 4, 5]  – залишок як список

*init, last = (1, 2, 3, 4, 5)
print(last)        # 5

a, b, *middle, c = (1, 2, 3, 4, 5, 6)
print(a, middle, c)  # 1 [2, 3, 4, 5] 6

# Swap через кортеж – класичний Python-прийом
a, b = 10, 20
a, b = b, a        # обмін без тимчасової змінної
print(a, b)        # 20 10

# Функція може повертати кілька значень (насправді – кортеж)
def circle(r):
    import math
    area      = round(math.pi * r**2, 2)
    perimeter = round(2 * math.pi * r, 2)
    return area, perimeter   # повертає кортеж

a, p = circle(5)    # розпакування при отриманні
print(f'Площа: {a}, Периметр: {p}')

result = circle(5)  # або зберегти весь кортеж
print(result)       # (78.54, 31.42)
print(result[0])    # 78.54
```

Ключова практична перевага кортежу перед списком – можливість використання як ключ словника або елемент множини. Це пов'язано з хешованістю: незмінні об'єкти мають фіксований хеш, тому їх можна зберігати у хеш-таблицях.

```
# Кортеж як ключ словника (список – не може!)
grid = {}
grid[(0, 0)] = 'початок'
grid[(1, 2)] = 'точка A'
grid[(3, 4)] = 'точка B'

print(grid[(1, 2)])   # точка A

# Координати міст
city_coords = {
    ('Київ',   50.45, 30.52): 'столиця',
    ('Харків', 49.99, 36.23): 'місто',
}

# list – не хешований, не може бути ключем
# d[[1, 2]] = 'x'  # TypeError: unhashable type: 'list'

# ── Коли tuple, а коли list? ──

# tuple: координати, RGB-колір, дата – дані не змінюються
rgb_red   = (255, 0, 0)
date      = (2025, 9, 1)
point     = (3.0, 4.0)

# list: список студентів, оцінки – дані змінюються
students = ['Іван', 'Марія']
students.append('Олег')   # OK

# Перетворення між типами
lst = [1, 2, 3]
tpl = tuple(lst)    # list -> tuple
back = list(tpl)    # tuple -> list
```

Практичне правило:
Використовуй `tuple` *коли: дані не мають змінюватися, потрібен ключ словника, функція повертає кілька значень, дані логічно пов'язані (координата, RGB).*
Використовуй `list` *коли: потрібно додавати, видаляти або змінювати елементи.*

Частина ІІ. Словники (dict)

Словник (dict) зберігає пари ключ-значення. Ключі мають бути унікальними та незмінними (рядки, числа, кортежі). Значення – будь-якого типу. Словник – одна з найуживаніших структур у Python: контакти, налаштування, бази даних, JSON-відповіді від API – все це природно зберігається як dict.

```
# Способи створення
empty  = {}
grades = {'Математика': 90, 'Фізика': 85, 'Python': 95}
person = dict(name='Іван', age=21, city='Київ')
keys   = dict.fromkeys(['x', 'y', 'z'], 0)  # {'x':0,'y':0,'z':0}
```

```
print(len(grades))   # 3  - кількість пар
print(type(grades))  # <class 'dict'>

# Доступ до значень
print(grades['Математика'])    # 90
# print(grades['Хімія'])       # KeyError - ключ відсутній!

# .get() - безпечний доступ, без KeyError
print(grades.get('Хімія'))         # None
print(grades.get('Хімія', 0))      # 0  - значення за замовчуванням

# Перевірка наявності ключа
print('Python' in grades)      # True
print('Хімія'  in grades)      # False

# Додавання та зміна
grades['Хімія']      = 78     # новий ключ
grades['Математика'] = 92     # зміна існуючого

# .setdefault() - додає ключ тільки якщо його немає
grades.setdefault('Англійська', 80)  # додає
grades.setdefault('Python', 99)      # не змінює - вже є!
print(grades['Python'])  # 95  - не 99

# Видалення
del grades['Хімія']                  # видалити ключ
score = grades.pop('Фізика')         # видалити і повернути
print(score)                         # 85
last = grades.popitem()              # видалити останню пару
```

Ітерація та методи словника

Метод / Синтаксис	**Опис / Приклад**
`d.keys()`	Всі ключі: `dict_keys(['Математика', ...])`
`d.values()`	Всі значення: `dict_values([90, 85, ...])`
`d.items()`	Всі пари: `dict_items([('Математика', 90), ...])`
`d.get(k, def)`	Значення або `default` якщо ключ відсутній
`d.setdefault(k,v)`	Повертає або встановлює значення
`d.update(other)`	Додає/перезаписує пари з іншого словника
`d.pop(k)`	Видаляє і повертає значення за ключем
`d.copy()`	Поверхнева копія словника
`d.clear()`	Видаляє всі пари

```
student = {'name': 'Іван', 'math': 85, 'physics': 90, 'cs': 95}

# Ітерація по ключах (за замовчуванням)
for key in student:
    print(key)

# Ітерація по значеннях
for value in student.values():
```

```
    print(value)

# Ітерація по парах – найчастіше використовується
for key, value in student.items():
    print(f'{key}: {value}')

# Об'єднання словників
a = {'x': 1, 'y': 2}
b = {'y': 10, 'z': 3}

merged = {**a, **b}    # {'x':1, 'y':10, 'z':3}  – b перезаписує a
a.update(b)            # оновлює a на місці
merged2 = a | b        # Python 3.9+: оператор |

# Сортування словника за значенням
scores = {'Іван': 85, 'Марія': 92, 'Олег': 78}
sorted_by_score = dict(sorted(scores.items(),
                              key=lambda x: x[1],
                              reverse=True))
print(sorted_by_score)
# {'Марія': 92, 'Іван': 85, 'Олег': 78}
```

Генератор словника (dict comprehension) – компактний синтаксис для створення нового словника. Аналогічний до спискового включення, але у фігурних дужках і з двокрапкою між ключем та значенням.

```
# {ключ: значення for змінна in послідовність}

# Квадрати чисел
squares = {x: x**2 for x in range(1, 6)}
print(squares)   # {1:1, 2:4, 3:9, 4:16, 5:25}

# Тільки предмети з оцінкою >= 90
student = {'math': 85, 'physics': 90, 'cs': 95, 'english': 78}
high = {k: v for k, v in student.items() if v >= 90}
print(high)   # {'physics': 90, 'cs': 95}

# Інверсія словника: ключ ↔ значення
original = {'a': 1, 'b': 2, 'c': 3}
inverted = {v: k for k, v in original.items()}
print(inverted)   # {1: 'a', 2: 'b', 3: 'c'}

# З двох списків
keys   = ['name', 'age', 'city']
values = ['Іван', 21, 'Київ']
person = {k: v for k, v in zip(keys, values)}
print(person)   # {'name': 'Іван', 'age': 21, 'city': 'Київ'}

# Підрахунок кількості символів
text     = 'hello'
char_cnt = {ch: text.count(ch) for ch in set(text)}
print(char_cnt)   # {'h':1, 'e':1, 'l':2, 'o':1}  (порядок може відрізнятись)
```

Значенням словника може бути будь-який тип, у тому числі інший словник. Вкладені словники використовуються для зберігання структурованих даних: картки студентів, налаштування програми, JSON-дані.

```python
# Вкладений словник: студенти університету
university = {
    'Іван': {
        'year':   2,
        'gpa':    4.2,
        'courses': ['Математика', 'Python', 'Фізика'],
    },
    'Марія': {
        'year':   3,
        'gpa':    4.8,
        'courses': ['AI', 'Python', 'Статистика'],
    },
}

# Доступ до вкладених значень
print(university['Іван']['gpa'])           # 4.2
print(university['Марія']['courses'][0])   # AI

# Безпечний доступ через .get() при вкладених ключах
gpa = university.get('Олег', {}).get('gpa', 0)
print(gpa)   # 0  (Олег відсутній, але помилки немає)

# Ітерація по вкладеному словнику
for name, info in university.items():
    print(f'{name}: курс {info["year"]}, GPA {info["gpa"]}')

# Додавання нового студента
university['Олег'] = {'year': 1, 'gpa': 3.5, 'courses': ['Python']}

# Оновлення вкладеного значення
university['Іван']['gpa'] = 4.5
university['Іван']['courses'].append('Алгоритми')

# Підрахунок середнього GPA через dict comprehension
avg_gpas = {name: info['gpa'] for name, info in university.items()}
avg = sum(avg_gpas.values()) / len(avg_gpas)
print(f'Середній GPA: {avg:.2f}')
```

Множини (set та frozenset)

Множина (set) – це змінна невпорядкована колекція унікальних елементів. Вона автоматично видаляє дублікати і забезпечує дуже швидку (O(1)) перевірку входження. Саме для цих двох задач – видалення дублікатів і швидка перевірка – множина є найкращим вибором.

```python
# Створення множин
s1 = {1, 2, 3, 4, 5}
s2 = set([1, 2, 2, 3, 3, 3])   # дублікати видаляються
print(s2)                       # {1, 2, 3}

s3 = set('hello')               # {'h','e','l','o'}  – унікальні символи

# УВАГА: {} – це порожній СЛОВНИК, не множина!
empty_dict = {}
```

```python
empty_set  = set()   # правильний порожній set
print(type(empty_dict))   # <class 'dict'>
print(type(empty_set))    # <class 'set'>

# Перевірка входження – O(1), набагато швидше ніж у списку
forbidden = {'spam', 'hack', 'bot'}
username  = 'hack'
if username in forbidden:
    print('Заборонене ім\'я!')

# Методи додавання та видалення
s = {1, 2, 3}
s.add(4)          # {1, 2, 3, 4}
s.add(2)          # дублікат ігнорується: {1, 2, 3, 4}
s.remove(1)       # {2, 3, 4}  – KeyError якщо немає
s.discard(99)     # нічого не відбувається, якщо немає
popped = s.pop()  # видалити ДОВІЛЬНИЙ елемент
s.clear()         # порожня множина: set()

print(len({1, 2, 3}))     # 3
print(3 in {1, 2, 3})     # True
```

Множини підтримують математичні операції теорії множин: об'єднання, перетин, різниця та симетрична різниця. Це дозволяє лаконічно вирішувати задачі порівняння колекцій.

```python
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7}

# Об'єднання: є хоча б в одній множині
print(a | b)           # {1, 2, 3, 4, 5, 6, 7}
print(a.union(b))

# Перетин: є в обох множинах
print(a & b)           # {3, 4, 5}
print(a.intersection(b))

# Різниця: є в a, але не в b
print(a - b)           # {1, 2}
print(a.difference(b))

# Симетрична різниця: є в одній, але не в обох
print(a ^ b)           # {1, 2, 6, 7}
print(a.symmetric_difference(b))

# Відношення між множинами
x = {1, 2}
y = {1, 2, 3, 4}
print(x.issubset(y))     # True  – x є підмножиною y
print(y.issuperset(x))   # True  – y містить x
print(x.isdisjoint({5, 6}))  # True  – нема спільних

# Оператори < і <=
print(x < y)   # True  – строга підмножина
print(x <= y)  # True  – підмножина або рівні
print(x == {2, 1})  # True  – порядок не важливий
```

Практичне застосування множин

```python
# ── 1. Видалення дублікатів зі списку ──
words   = ['apple','banana','apple','cherry','banana','date']
unique  = list(set(words))
print(unique)   # (порядок не гарантований)

# ── 2. Спільні елементи двох списків ──
list_a = [1, 2, 3, 4, 5, 6]
list_b = [4, 5, 6, 7, 8, 9]
common = set(list_a) & set(list_b)
print(common)   # {4, 5, 6}

# ── 3. Унікальні слова у тексті ──
text   = 'python is great python is easy python is fun'
unique_words = set(text.split())
print(f'Унікальних слів: {len(unique_words)}')
print(sorted(unique_words))   # алфавітний порядок

# ── 4. Перевірка на дублікати ──
def has_duplicates(lst):
    return len(lst) != len(set(lst))

print(has_duplicates([1, 2, 3, 4]))  # False
print(has_duplicates([1, 2, 2, 3]))  # True

# ── 5. frozenset – незмінна множина (як ключ словника) ──
fs = frozenset([1, 2, 3])
# fs.add(4)  # AttributeError – незмінна!

# frozenset може бути ключем словника
graph = {}
graph[frozenset({'A', 'B'})] = 'ребро AB'
graph[frozenset({'B', 'C'})] = 'ребро BC'
print(graph[frozenset({'A', 'B'})])  # ребро AB

# ── 6. Генератор множини ──
vowels = {c for c in 'programming' if c in 'aeiou'}
print(vowels)   # {'o', 'a', 'i'}
```

Завдання на лабораторну роботу

3.1. Виконайте серію вправ з кортежами:

- Створіть кортеж із 5 назв міст. Виведіть перше та останнє місто, виведіть у зворотньому порядку, знайдіть довжину кортежу.
- Напишіть функцію `minmax(numbers)`, що приймає список чисел і повертає кортеж (мінімум, максимум). Розпакуйте результат у змінні `lo, hi`.
- Використайте розпакування з *: для кортежу (1, 2, 3, 4, 5) отримайте першу, останню і середню частину у три змінні.
- Поміняйте місцями значення трьох змінних `a, b, c` через паралельне присвоєння (без тимчасових змінних): `a, b, c = b, c, a`.
- Поясніть чому (5) – це `int`, а (5,) – це `tuple`. Продемонструйте обидва варіанти.

3.2. Створіть словник `grades` – оцінки студента з 6 предметів. Виконайте:
- Виведіть всі предмети (ключі) та всі оцінки (значення).
- Виведіть предмет і оцінку через `items()` у форматі 'Предмет: оцінка'.
- Знайдіть предмет з найвищою оцінкою (`max() з key=`).
- Обчисліть середню оцінку (`sum + len`).
- Додайте новий предмет, змініть одну оцінку, видаліть один предмет через `pop()`.
- Відсортуйте словник за оцінкою (від найвищої до найнижчої).

3.3. Задайте три множини: `students_math, students_physics, students_cs` – прізвища студентів (мінімум по 5, деякі спільні). Виконайте:
- Знайдіть і виведіть студентів, що відвідують усі три предмети.
- Знайдіть студентів, що відвідують хоча б один предмет.
- Знайдіть студентів математики, що НЕ відвідують фізику.
- Перевірте чи є `students_cs` підмножиною `students_math`.
- Знайдіть студентів, що відвідують рівно два предмети.
- Видаліть дублікати зі списку [1,2,2,3,3,3,4] і виведіть результат як список.

3.4. Виконайте наступні задачі:
- Через dict comprehension: створіть словник {число: куб числа} для чисел від 1 до 10.
- Через dict comprehension з умовою: зі словника {'яблуко': 35, 'банан': 22, 'ківі': 55, 'слива': 18} відберіть тільки товари дешевше 40 грн.
- Задайте список студентів і їх середніх оцінок у форматі [{'name': ..., 'gpa': ...}, ...] (мінімум 5 студентів). Через dict comprehension побудуйте словник середніх оцінок студентів {name: gpa}.
- Реалізуйте вкладений словник 'телефонна книга' (ім'я → {phone, email, city}). Виведіть усі контакти з Києва. Додайте новий контакт. Безпечно отримайте email для відсутнього контакту через .get().

3.5. Напишіть програму для аналізу тексту (задайте рядок з мінімум 30 словами):
- Підрахуйте кількість входжень кожного слова (без урахування регістру) вручну через словник і цикл for (без Counter з модуля collections)
- Знайдіть 3 найуживаніших слова через sorted() з key=
- Підрахуйте кількість входжень кожного символу (тільки букви)
- Побудуйте 'зворотній індекс': для кожного унікального слова – список позицій (індексів) у тексті де воно зустрічається
- Виведіть кількість унікальних слів та відсоток унікальних серед усіх

3.6. Розв'яжіть такі задачі, комбінуючи dict, set та tuple:
- Задайте словник розкладу занять: {день_тижня: [(час, предмет, аудиторія), ...]}. Знайдіть усі унікальні аудиторії через множину. Виведіть заняття в понеділок. Знайдіть день де найбільше занять.
- Дано список транзакцій – кортежів (дата, категорія, сума). Мінімум 15 транзакцій, 4 категорії. Через dict підрахуйте суму витрат по кожній категорії. Через set знайдіть унікальні дати. Знайдіть категорію з найбільшими витратами.

3.7. Реалізуйте просту систему реєстрації студентів лише через dict, set та tuple (без класів). Структура даних: словник students де ключ – student_id (рядок 'ST-001', 'ST-002' тощо),

значення – кортеж (name, year, gpa, frozenset_of_courses).
Реалізуйте функції (кожна приймає словник students як перший аргумент):

- add_student(students, name, year, gpa, courses) → оновлює словник, автоматично генерує id.
- find_by_name(students, name) → повертає id і дані студента або None.
- top_students(students, n=3) → список n кортежів (name, gpa) відсортованих за GPA.
- students_in_course(students, course_name) → множина імен студентів на курсі.
- common_courses(students, id1, id2) → множина спільних курсів двох студентів.
- stats(students) → словник: {'total': N, 'avg_gpa': X, 'unique_courses': set}.

Наповніть систему мінімум 6 студентами з різними курсами. Виведіть результат кожної функції. Покажіть що frozenset курсів можна використовувати як ключ (наприклад, знайти студентів з однаковим набором курсів).

Контрольні запитання

1. Яка принципова різниця між tuple і list? Чому tuple може бути ключем словника, а list – ні?
2. Що означає запис (42,) і чим він відрізняється від (42)? Як Python відрізняє кортеж з одним елементом від виразу у дужках?
3. Що таке розпакування кортежу? Що виконає команда a, *b, c = (1, 2, 3, 4, 5)? Які значення матимуть a, b, c?
4. Яка різниця між dict[key] і dict.get(key)? Коли кожен підхід є кращим?
5. Що повертають методи .keys(), .values() та .items()? Як ітерувати словник, щоб отримати і ключ, і значення одночасно?
6. Що таке dict comprehension? Запишіть вираз, що зі списку слів ['apple','banana','cherry'] створює словник {слово: довжина_слова}.
7. Чому {} – це порожній словник, а не множина? Як правильно створити порожню множину?
8. Поясніть різницю між операціями & і | для множин. Що таке симетрична різниця і коли вона корисна?
9. Чим frozenset відрізняється від set? Наведіть приклад використання frozenset як ключа словника.
10. Назвіть три практичні задачі, де множина є кращим вибором ніж список. Поясніть чому перевірка входження у множині O(1), а в списку O(n).

Лабораторна робота №4.

Функції

Мета роботи: вивчити синтаксис визначення та виклику функцій у Python, види параметрів; засвоїти концепцію областей видимості змінних (LEGB-правило); ознайомитись з техніками функціонального програмування у Python, лямбда-функціями, замиканнями (closures) та декораторами.

Теоретичні відомості:

Функція – це іменований блок коду, який виконує певну задачу і може бути викликаний багаторазово. Функції дозволяють уникнути дублювання коду, розбити програму на логічні частини та полегшити тестування.

```
# Найпростіша функція без параметрів і без повернення
def greet():
    print('Привіт, світ!')

greet()            # Виклик функції
greet()            # Можна викликати скільки завгодно разів

# Функція з параметром
def greet_user(name):
    print(f'Привіт, {name}!')

greet_user('Олена')
greet_user('Іван')

# Функція з поверненням результату (return)
def square(x):
    return x ** 2

result = square(7)
print(result)      # 49
print(square(5))   # 25  - можна прямо у print()

# Функція є об'єктом – її можна присвоїти змінній
func = square
print(func(4))     # 16
```

Зверніть увагу!

Функція без явного `return` *повертає* `None`.

```
def f():
    print('ok')

result = f()
print(result)
```

```
ok
None
```

Тому завжди явно пишіть `return`*, якщо функція має щось повертати.*

Параметри – це імена змінних при оголошенні функції. Аргументи – це значення, що передаються в функцію при виклику. Python підтримує кілька видів параметрів.

Позиційні та іменовані аргументи

```
def describe_pet(name, animal, age):
    print(f'{name} - це {animal}, вік {age} років')
# Позиційні - порядок має значення
describe_pet('Барсик', 'кіт', 3)

# Іменовані (keyword) - порядок не важливий
describe_pet(age=5, name='Рекс', animal='пес')

# Можна змішувати: позиційні ЗАВЖДИ йдуть першими
describe_pet('Чоко', age=2, animal='папуга')
```

Значення за замовчуванням (default values)

```
# Параметр зі значенням за замовчуванням
def power(base, exp=2):           # exp=2 за замовчуванням
    return base ** exp

print(power(5))       # 25   (exp=2 за замовчуванням)
print(power(5, 3))    # 125  (exp=3 - перевизначено)
print(power(2, 10))   # 1024

# Реальний приклад: функція вітання з мовою за замовчуванням
def welcome(name, lang='uk'):
    messages = {
        'uk': f'Привіт, {name}!',
        'en': f'Hello, {name}!',
        'de': f'Hallo, {name}!',
    }
    return messages.get(lang, f'Hi, {name}!')

print(welcome('Олена'))           # Привіт, Олена!
print(welcome('Anna', 'en'))      # Hello, Anna!
```

Поширена помилка!

НІКОЛИ не використовуйте змінюваний об'єкт як значення за замовчуванням!

```
def bad(lst=[]):
    lst.append(1)
    return lst   # НЕБЕЗПЕЧНО!
bad()  # [1]       bad()  # [1, 1]  - об'єкт спільний для всіх викликів!
bad()
bad()
bad()
print(bad())
```

```
[1, 1, 1, 1, 1]
```

```
# ПРАВИЛЬНО!
def good(lst=None):
    if lst is None: lst = []
```

*args – довільна кількість позиційних аргументів

```python
# *args збирає всі зайві позиційні аргументи у кортеж
def total(*args):
    print(f'Отримано аргументів: {len(args)}')
    print(f'Значення: {args}')
    return sum(args)

print(total(1, 2, 3))          # 6
print(total(10, 20, 30, 40))   # 100
print(total(5))                # 5

# Змішування: фіксований параметр + *args
def log(level, *messages):
    for msg in messages:
        print(f'[{level}] {msg}')

log('INFO', 'Програма запущена', 'Читання файлу')
log('ERROR', 'Щось пішло не так')
```

**kwargs – довільна кількість іменованих аргументів

```python
# **kwargs збирає іменовані аргументи у словник
def show_info(**kwargs):
    for key, value in kwargs.items():
        print(f'  {key}: {value}')

show_info(name='Іван', age=21, city='Київ')
# name: Іван
# age: 21
# city: Київ

# Практичний приклад: функція створення HTML-тегу
def make_tag(tag, content, **attrs):
    attr_str = ' '.join(f'{k}="{v}"' for k, v in attrs.items())
    if attr_str:
        return f'<{tag} {attr_str}>{content}</{tag}>'
    return f'<{tag}>{content}</{tag}>'

print(make_tag('p', 'Текст'))
# <p>Текст</p>
print(make_tag('a', 'Клік', href='https://example.com', class_='link'))
# <a href="https://example.com" class_="link">Клік</a>
```

Повний порядок параметрів

```python
# Правило: позиційні → *args → keyword-only → **kwargs
def full_example(pos1, pos2, *args, kw_only, **kwargs):
    print(f'Позиційні: {pos1}, {pos2}')
    print(f'*args: {args}')
    print(f'kw_only: {kw_only}')
    print(f'**kwargs: {kwargs}')

full_example(1, 2, 3, 4, 5, kw_only='обов\'язковий', extra=99)
# Позиційні: 1, 2
# *args: (3, 4, 5)
# kw_only: обов'язковий
# **kwargs: {'extra': 99}
```

```
# Розпакування при виклику: * і **
nums = [3, 7]
print(power(*nums))        # = power(3, 7)  → 2187

params = {'name': 'Марко', 'lang': 'en'}
print(welcome(**params))   # = welcome(name='Марко', lang='en')
```

Функція може повернути будь-який об'єкт Python. Якщо потрібно повернути кілька значень, Python автоматично упаковує їх у кортеж.

```
# Повернення одного значення
def circle_area(r):
    import math
    return math.pi * r ** 2

# Повернення кількох значень (насправді – кортеж)
def circle_info(r):
    import math
    area      = math.pi * r ** 2
    perimeter = 2 * math.pi * r
    return area, perimeter          # повертає кортеж (area, perimeter)

a, p = circle_info(5)               # розпакування кортежу
print(f'Площа: {a:.2f}')
print(f'Периметр: {p:.2f}')

result = circle_info(5)             # або зберегти як кортеж
print(result)                       # (78.53..., 31.41...)

# Ранній вихід з функції через return
def safe_divide(a, b):
    if b == 0:
        return None                 # Ранній вихід
    return a / b

print(safe_divide(10, 2))  # 5.0
print(safe_divide(10, 0))  # None
```

Python визначає значення імені за правилом LEGB: Python шукає змінну послідовно у чотирьох «шарах» від внутрішнього до зовнішнього і зупиняється на першому знайденому.

Синтаксис	Опис / Приклад
`L – Local`	Локальний блок: змінні, визначені всередині поточної функції
`E – Enclosing`	Охоплююча функція: змінні зовнішньої функції (при вкладенні)
`G – Global`	Модульний рівень: змінні, визначені на рівні файлу/модуля
`B – Built-in`	Вбудований простір: `len`, `print`, `range`, `int` та інші

```
x = 'глобальна'         # G – глобальна змінна

def outer():
    x = 'охоплююча'  # E – у зовнішній функції

    def inner():
        x = 'локальна' # L – локальна змінна inner()
        print(x)       # 'локальна'  (знайдено у L)
```

```
    inner()
    print(x)           # 'об\'ємлююча'  (знайдено у E)

outer()
print(x)               # 'глобальна'  (знайдено у G)
```

Ключове слово global дозволяє зробити змінну глобальною

```
counter = 0            # глобальна змінна

def increment():
    global counter     # оголошуємо: працюємо з глобальною
    counter += 1

def get_counter():
    return counter     # читати глобальну можна без global
increment()
increment()
increment()
print(get_counter())   # 3

# Без global – помилка UnboundLocalError!
def bad_increment():
    # counter += 1   # ПОМИЛКА: Python бачить присвоєння
    pass             # і вважає counter локальною (ще не ініціалізованою)
```

Ключове слово `nonlocal`

```
# nonlocal – для змінної охоплюючої (не глобальної) функції
def make_counter():
    count = 0

    def increment():
        nonlocal count     # змінюємо змінну з outer-функції
        count += 1
        return count

    return increment

counter = make_counter()
print(counter())   # 1
print(counter())   # 2
print(counter())   # 3

# Два незалежних лічильники
c1 = make_counter()
c2 = make_counter()
c1(); c1()         # c1 → 2
c2()               # c2 → 1  (незалежний!)
```

Docstring – рядок документації одразу після `def`. Він пояснює, що робить функція, і доступний через `help()` та атрибут `__doc__`.

```
def calculate_bmi(weight: float, height: float) -> float:
    '''
    Обчислює індекс маси тіла (ІМТ).
```

```
    Параметри:
        weight (float): вага у кілограмах
        height (float): зріст у метрах

    Повертає:
        float: значення ІМТ, округлене до 2 знаків

    Приклад:
        >>> calculate_bmi(70, 1.75)
        22.86
    '''
    return round(weight / height ** 2, 2)

# Анотації типів – підказки (не примусово!)
def greet(name: str, times: int = 1) -> str:
    return (f'Привіт, {name}! ' * times).strip()

print(greet('Іван', 3))
# Привіт, Іван! Привіт, Іван! Привіт, Іван!

# Виведення документації
help(calculate_bmi)
```

Рекурсія – це техніка, коли функція викликає сама себе. Обов'язкові умови: базовий випадок (умова зупинки) і крок наближення до нього.

```
# Класичний приклад: факторіал
def factorial(n):
    if n == 0 or n == 1:   # базовий випадок
        return 1
    return n * factorial(n - 1)   # рекурсивний виклик

print(factorial(5))   # 120
print(factorial(10))  # 3628800

# Числа Фібоначчі
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)

print([fib(i) for i in range(10)])
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

# Сума цифр числа рекурсивно
def digit_sum(n):
    n = abs(n)
    if n < 10:
        return n
    return n % 10 + digit_sum(n // 10)

print(digit_sum(12345))   # 15
```

Обмеження рекурсії!

Python обмежує глибину рекурсії до ~1000 викликів (`sys.getrecursionlimit()`*).*
Для великих n рекурсія Фібоначчі дуже повільна – краще використовуйте цикл. При надмірній глибині буде кинуто виняток `RecursionError`.

Лямбда (lambda) – це коротка анонімна функція в одному рядку. Вона може мати будь-яку кількість параметрів, але лише один вираз, результат якого автоматично повертається.

```
# Звичайна функція і її лямбда-еквівалент
def square(x):
    return x ** 2

square_l = lambda x: x ** 2

print(square(7))      # 49
print(square_l(7))    # 49  - однаково

# Лямбда з кількома параметрами
add  = lambda x, y: x + y
full = lambda first, last: f'{first} {last}'

print(add(3, 4))              # 7
print(full('Іван', 'Коваль')) # Іван Коваль

# Лямбда з умовою (тернарний оператор)
classify = lambda n: 'парне' if n % 2 == 0 else 'непарне'
print(classify(6))    # парне
print(classify(7))    # непарне
```

Лямбда у функціях вищого порядку. Найбільша цінність лямбда – використання як аргумент у функціях, що приймають іншу функцію: `sorted()`, `map()`, `filter()`, `reduce()`.

```
# sorted() з лямбда як key
students = [('Марія', 95), ('Іван', 78), ('Олег', 88), ('Ліна', 95)]

# Сортування за балом (за спаданням)
by_score = sorted(students, key=lambda s: s[1], reverse=True)
print(by_score)
# [('Марія', 95), ('Ліна', 95), ('Олег', 88), ('Іван', 78)]

# Сортування за двома критеріями: спочатку за балом (спад), потім за ім'ям (зріст)
by_two = sorted(students, key=lambda s: (-s[1], s[0]))
print(by_two)
# [('Ліна', 95), ('Марія', 95), ('Олег', 88), ('Іван', 78)]

# map() - застосувати функцію до кожного елементу
nums = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x ** 2, nums))
print(squares)        # [1, 4, 9, 16, 25]

# filter() - відфільтрувати елементи
evens = list(filter(lambda x: x % 2 == 0, range(10)))
print(evens)          # [0, 2, 4, 6, 8]

# reduce() - згортка списку в одне значення
from functools import reduce
product = reduce(lambda acc, x: acc * x, [1, 2, 3, 4, 5])
print(product)        # 120  (1*2*3*4*5)
```

Коли лямбда, а коли def?

Лямбда підходить для простих одноразових функцій (особливо як аргумент).
def підходить для складної логіки, рекурсії, докстрінгів, повторного використання.
PEP8 радить НЕ присвоювати лямбду змінній: краще написати повноцінний def.

Замикання (Closures) виникає, коли вкладена функція «запам'ятовує» змінні зовнішньої функції навіть після того, як зовнішня функція завершила виконання. Вкладена функція несе з собою своє оточення.

Три умови замикання: 1) вкладена функція; 2) вона посилається на змінну зовнішньої функції; 3) зовнішня функція повертає вкладену

```python
# Базовий приклад замикання
def make_multiplier(factor):
    '''Повертає функцію, що множить на factor'''
    def multiplier(x):                # замикається на factor
        return x * factor
    return multiplier                 # повертаємо функцію (без дужок!)

double = make_multiplier(2)
triple = make_multiplier(3)

print(double(5))   # 10
print(triple(5))   # 15
print(double(8))   # 16

# Кожне замикання зберігає СВОЄ значення factor
print(double.__closure__[0].cell_contents)  # 2
print(triple.__closure__[0].cell_contents)  # 3
```

```python
# Замикання як генератор функцій – практичний приклад
def make_power(exp):
    return lambda x: x ** exp

square = make_power(2)
cube   = make_power(3)
quad   = make_power(4)

print([square(i) for i in range(1, 6)])  # [1, 4, 9, 16, 25]
print([cube(i)   for i in range(1, 6)])  # [1, 8, 27, 64, 125]

# Замикання зі збереженням стану (аналог об'єкта)
def make_bank_account(initial=0):
    balance = [initial]               # список, бо nonlocal не потрібен

    def deposit(amount):
        balance[0] += amount
        return balance[0]

    def withdraw(amount):
        if amount > balance[0]:
            return 'Недостатньо коштів'
        balance[0] -= amount
        return balance[0]

    def get_balance():
        return balance[0]

    return deposit, withdraw, get_balance

dep, wdr, bal = make_bank_account(1000)
print(dep(500))    # 1500
print(wdr(200))    # 1300
print(bal())       # 1300
```

Декоратор – це функція, яка приймає іншу функцію як аргумент, розширює її поведінку і повертає нову (модифіковану) функцію. Декоратори не змінюють оригінальну функцію, а «огортають» її.

Базовий декоратор

```python
# Крок 1: функція-обгортка (wrapper)
def my_decorator(func):
    def wrapper(*args, **kwargs):    # приймає будь-які аргументи
        print(f'--- Виклик {func.__name__} ---')
        result = func(*args, **kwargs)  # виклик оригінальної функції
        print(f'--- Завершено ---')
        return result
    return wrapper

# Крок 2: застосування через @
@my_decorator
def add(a, b):
    return a + b

# Це еквівалентно:  add = my_decorator(add)

print(add(3, 4))
# --- Виклик add ---
# --- Завершено ---
# 7
```

Декоратор із збереженням метаданих

```python
from functools import wraps

def log_calls(func):
    @wraps(func)              # зберігає __name__, __doc__
    def wrapper(*args, **kwargs):
        args_repr = ', '.join(repr(a) for a in args)
        print(f'Виклик: {func.__name__}({args_repr})')
        result = func(*args, **kwargs)
        print(f'Результат: {result!r}')
        return result
    return wrapper

@log_calls
def multiply(x, y):
    '''Множить два числа'''
    return x * y

multiply(4, 7)
# Виклик: multiply(4, 7)
# Результат: 28

print(multiply.__name__)  # multiply  (не wrapper!)
print(multiply.__doc__)   # Множить два числа
```

Практичні декоратори

```python
import time
from functools import wraps

# — Декоратор вимірювання часу виконання —
def timer(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        elapsed = time.perf_counter() - start
        print(f'{func.__name__} виконано за {elapsed:.4f} с')
        return result
    return wrapper

@timer
def slow_sum(n):
    return sum(range(n))

slow_sum(10_000_000)  # slow_sum виконано за 0.3xx с

# — Декоратор кешування результатів —
def memoize(func):
    cache = {}
    @wraps(func)
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

@memoize
def fib(n):
    if n <= 1: return n
    return fib(n-1) + fib(n-2)

print(fib(40))   # Блискавично! (без кешу – хвилини)
```

Декоратори з параметрами

```python
# Декоратор з параметром – потрібен ще один рівень вкладення
def repeat(times):
    '''Повторює виклик функції times разів'''
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            result = None
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def say_hello(name):
    print(f'Привіт, {name}!')

say_hello('Олена')
# Привіт, Олена!
```

```
# Привіт, Олена!
# Привіт, Олена!

# Кілька декораторів застосовуються знизу вгору
@log_calls
@timer
def compute(n):
    return sum(i**2 for i in range(n))

compute(1000)
# Спочатку timer огортає compute,
# потім log_calls огортає timer(compute)
```

У Python функції є об'єктами першого класу: їх можна зберігати у змінних, передавати як аргументи, повертати з інших функцій і зберігати у структурах даних.

```
# Зберігання функцій у словнику – «диспетчер»
def add(a, b):      return a + b
def sub(a, b):      return a - b
def mul(a, b):      return a * b
def div(a, b):      return a / b if b != 0 else 'Ділення на 0'

operations = {'+': add, '-': sub, '*': mul, '/': div}

op = input('Операція (+,-,*,/): ')
a  = float(input('a: '))
b  = float(input('b: '))

if op in operations:
    print(f'Результат: {operations[op](a, b)}')
else:
    print('Невідома операція')

# Функція вищого порядку – приймає і повертає функцію
def apply_twice(func, value):
    return func(func(value))

print(apply_twice(lambda x: x * 2, 3))   # 12  (3→6→12)
print(apply_twice(lambda x: x + 5, 10))  # 20  (10→15→20)

# Конвеєр (pipeline) функцій
def pipeline(*funcs):
    def process(value):
        for f in funcs:
            value = f(value)
        return value
    return process

clean = pipeline(
    str.strip,
    str.lower,
    lambda s: s.replace(' ', '_')
)

print(clean('  Hello World  '))  # hello_world
```

Завдання на лабораторну роботу

5.1. Базові функції. Напишіть три функції:
- `is_even(n)` – повертає `True`, якщо `n` парне;
- `max_of_three(a, b, c)` – повертає найбільше з трьох чисел;
- `power(base, exp=2)` – піднесення до степеня (з параметром за замовчуванням).

Для кожної функції напишіть мінімум 3 тестових виклики з виведенням результатів.

5.2. Функції з *args. Реалізуйте наступні функції:
- `sum_list(numbers)` – повертає суму елементів списку довільної довжини;
- `stats(*numbers)` – повертає словник `{'min': ..., 'max': ..., 'sum': ..., 'avg': ...}` з усіх числових аргументів, що були передані в функцію;

Продемонструйте виклики з різною кількістю аргументів.

5.3. Рекурсивні функції. Реалізуйте такі рекурсивні функції:
- `power(base, exp)` – піднесення до степеня без використання **;
- `factorial(number)` – визначення факторіала числа `number`.

Для кожної функції продемонструйте не менше 3 викликів.

5.4. Лямбда і функції вищого порядку. Виконайте всі завдання, використовуючи виключно `lambda`, `map()`, `filter()`, `sorted()`:
- зі списку чисел від -10 до 10 відфільтруйте тільки ті, що кратні 3;
- перетворіть список температур в градусах Цельсія [0, 20, 37, 100] у градуси Кельвіна;
- відсортуйте список рядків за останньою літерою слова;
- зі списку словників `[{'name': 'Іван', 'age': 22}, ...]` відфільтруйте тих, кому `>= 21`, відсортуйте за іменем.

5.5. Комплексна задача: функціональна обробка даних. Дано список студентів у вигляді словників з полями `name`, `grades` (список оцінок від 0 до 100), `year` (рік навчання). Задайте мінімум 8 студентів різних курсів.

Реалізуйте такі функції (без використання зовнішніх бібліотек):
- `average(grades)` – середня оцінка з точністю до 2 знаків (рекурсивно або через reduce);
- `classify(avg)` – повертає категорію: 'відмінник' (>=90), 'хорошист' (>=75), 'задовільно' (>=60), 'незараховано' (<60)
- `get_top(students, n=3)` – повертає n найкращих студентів (через `sorted` + `lambda`)
- `group_by_year(students)` – повертає словник `{рік: [список студентів]}`
- `report(students)` – декорована `@timer` функція, що виводить повну статистику: загальне зведення, топ-3, розбивка по роках, розподіл за категоріями

Вся логіка обробки має використовувати `map()`, `filter()`, `sorted()`, `lambda` і замикання.

Контрольні запитання

1. Що таке функція у Python? Чому використання функцій покращує якість коду?
2. Яка різниця між параметром і аргументом? Поясніть на прикладі.
3. Що таке `*args` і `**kwargs`? Який порядок параметрів є обов'язковим у визначенні функції?
4. Що таке LEGB-правило? Опишіть кожен рівень і наведіть приклад, де Python знаходить змінну на кожному рівні.
5. Для чого потрібні ключові слова `global` та `nonlocal`? Чим вони відрізняються?
6. Що таке лямбда-функція? Коли доцільно використовувати `lambda`, а коли `def`? Наведіть приклад використання `lambda` у `sorted()`.
7. Що таке замикання (closure)? Назвіть три умови його виникнення та поясніть практичне застосування.
8. Що таке декоратор? Поясніть, що означає запис `@my_decorator` над визначенням функції.
9. Для чого потрібен `@wraps(func)` всередині декоратора? Що відбудеться, якщо його не використати?
10. Що таке функція вищого порядку? Наведіть три приклади вбудованих функцій Python, що є функціями вищого порядку.

Лабораторна робота №5.

Обробка помилок та логування

Мета роботи: Зрозуміти природу винятків (exceptions) у Python та відмінність між синтаксичними помилками і помилками часу виконання; навчитися перехоплювати та обробляти різні типи помилок за допомогою блоків try-except, використовувати блоки else та finally для контролю потоку виконання. Ознайомитися з ієрархією вбудованих винятків Python, навчитися свідомо генерувати помилки через raise та assert, а також створювати власні класи винятків для точного опису помилок у прикладних програмах. Засвоїти роботу з модулем logging: налаштування рівнів, форматів та обробників (handlers) для запису подій і помилок у консоль та файли.

Теоретичні відомості:

У Python існують два принципово різних типи помилок:

- Синтаксична помилка (SyntaxError) – виявляється до запуску програми. Це помилка у написанні коду: пропущені дужки, неправильні відступи тощо. Python взагалі не запускає програму.
- Виняток (Exception) – виникає під час виконання програми (runtime error). Код синтаксично правильний, але при виконанні щось пішло не так: ділення на нуль, звернення до відсутнього ключа, відкриття неіснуючого файлу.

```
# Синтаксична помилка – програма навіть не запуститься
# if x > 0    # SyntaxError: expected ':'
# print('hello'  # SyntaxError: '(' was never closed

# Виняток – виникає під час виконання
x = 10 / 0            # ZeroDivisionError
lst = [1, 2, 3]
print(lst[5])          # IndexError
int('abc')             # ValueError
open('nofile.txt')     # FileNotFoundError

# Без обробки – програма аварійно завершується і виводить traceback:
# Traceback (most recent call last):
#   File 'main.py', line 2, in <module>
#     x = 10 / 0
# ZeroDivisionError: division by zero
```

Блок try-except перехоплює виняток і дозволяє програмі продовжити роботу замість аварійного завершення.

```
# Базовий синтаксис
try:
    # Код, що може викинути виняток
    result = 10 / 0
except ZeroDivisionError:
    # Виконується якщо виник саме ZeroDivisionError
    print('Помилка: ділення на нуль!')

# Кілька except – обробка різних типів помилок
```

```python
def safe_convert(value):
    try:
        return int(value)
    except ValueError:
        print(f'Не вдалося перетворити "{value}" у число')
        return None
    except TypeError:
        print(f'Неправильний тип: {type(value).__name__}')
        return None

print(safe_convert('42'))    # 42
print(safe_convert('abc'))   # Не вдалося перетворити...
print(safe_convert(None))    # Неправильний тип...

# Отримати об'єкт винятку через 'as'
try:
    lst = [1, 2, 3]
    print(lst[10])
except IndexError as e:
    print(f'Помилка: {e}')         # list index out of range
    print(f'Тип: {type(e).__name__}')  # IndexError

# Кілька типів в одному except
try:
    x = int(input('Число: '))
    print(10 / x)
except (ValueError, ZeroDivisionError) as e:
    print(f'Введіть ненульове ціле число. ({e})')
```

Повна конструкція обробки винятків може містити чотири блоки. Кожен має своє призначення:

- `try` – код, що може спричинити виняток;
- `except` – виконується тільки якщо виняток було перехоплено;
- `else` – виконується тільки якщо виняток не було перехоплено;
- `finally` – виконується ЗАВЖДИ, незалежно від результату.

```python
def read_file(filename):
    f = None
    try:
        f = open(filename, 'r', encoding='utf-8')
        content = f.read()
        return content
    except FileNotFoundError:
        print(f'Файл "{filename}" не знайдено')
        return None
    except PermissionError:
        print(f'Немає прав доступу до "{filename}"')
        return None
    else:
        # Виконується тільки якщо відкриття і читання пройшли успішно
        print(f'Файл успішно прочитано: {len(content)} символів')
    finally:
        # Виконується ЗАВЖДИ – ресурс звільняється
        if f:
            f.close()
            print('Файл закрито')
```

```python
# else у try-except – корисний патерн валідації
while True:
    try:
        age = int(input('Введіть вік: '))
    except ValueError:
        print('Будь ласка, введіть ціле число')
    else:
        # Конвертація пройшла успішно – виходимо з циклу
        if 1 <= age <= 120:
            break
        print('Вік має бути від 1 до 120')

print(f'Ваш вік: {age}')
```

Правило: `finally` завжди виконується!

Навіть якщо в `try` або `except` є `return` – `finally` спрацює.
Навіть якщо виняток не оброблено – `finally` спрацює перед аварійною зупинкою програми.
Використовуйте `finally` для звільнення ресурсів (файли, з'єднання).
На практиці for файлів краще використовувати `with open(...) as f`.

Всі винятки Python успадковують від BaseException. Коли ви перехоплюєте батьківський клас – ви перехоплюєте і всі його підкласи.

Виняток	Опис і причина
`BaseException`	Кореневий вийняток. Рідко перехоплюється напряму.
`Exception`	Батьківський вийняток більшості програмних помилок.
`ArithmeticError`	Математичні помилки: `ZeroDivisionError`, `OverflowError`.
`ZeroDivisionError`	Ділення або взяття модуля на нуль: `10 / 0`
`ValueError`	Правильний тип, але неправильне значення: `int('abc')`
`TypeError`	Операція для неправильного типу: `'a' + 1`
`IndexError`	Індекс поза межами: `lst[100] для lst=[1,2,3]`
`KeyError`	Ключ відсутній у словнику: `d['missing']`
`AttributeError`	Об'єкт не має такого атрибута: `None.split()`
`FileNotFoundError`	Файл або директорія не існує
`PermissionError`	Немає прав доступу
`ImportError`	Модуль не знайдено або не вдалося імпортувати
`StopIteration`	Ітератор вичерпаний (`GeneratorExit` – підклас)
`KeyboardInterrupt`	Користувач натиснув Ctrl+C (не Exception!)

```python
# Батьківський клас перехоплює і дочірні
try:
    pass  # деякий код
except ArithmeticError:
    # Перехоплює ZeroDivisionError, OverflowError, FloatingPointError
    pass

# except Exception – 'catch all' для програмних помилок
```

```
# НЕ перехоплює KeyboardInterrupt, SystemExit
try:
    risky_code()
except Exception as e:
    print(f'Несподівана помилка: {type(e).__name__}: {e}')

# Завжди ловіть КОНКРЕТНИЙ тип – це найкраща практика!
# ПОГАНО:  except:          (ловить все, навіть Ctrl+C)
# ПОГАНО:  except Exception: (занадто широко)
# ДОБРЕ:   except ValueError, except FileNotFoundError тощо
```

Оператор `raise` дозволяє програмно генерувати виняток. Це корисно для валідації вхідних даних та сигналізації про некоректний стан програми.

```
# raise з явним типом і повідомленням
def set_age(age: int) -> None:
    if not isinstance(age, int):
        raise TypeError(f'Вік має бути int, отримано {type(age).__name__}')
    if age < 0 or age > 150:
        raise ValueError(f'Некоректний вік: {age}. Має бути 0–150.')
    print(f'Вік встановлено: {age}')

set_age(25)       # OK
# set_age(-5)     # ValueError: Некоректний вік: -5...
# set_age('old')  # TypeError: Вік має бути int...

# Перевикидання винятку (re-raise) – зберігає traceback
def safe_open(path):
    try:
        return open(path, encoding='utf-8')
    except FileNotFoundError:
        print(f'Логуємо: файл {path} не знайдено')
        raise           # повторно кидає той самий виняток

# raise ... from ... – ланцюжок помилок (chaining)
try:
    x = int('abc')
except ValueError as original:
    raise RuntimeError('Не вдалося обробити дані') from original
```

`assert` – перевірка умов у налагодженні

```
# assert умова, 'повідомлення'
# Кидає AssertionError якщо умова False

def divide(a, b):
    assert b != 0, 'Дільник не може бути нулем'
    return a / b

print(divide(10, 2))    # 5.0
# divide(10, 0)         # AssertionError: Дільник не може бути нулем

# assert корисний для перевірки передумов у функціях
def process_list(lst: list):
    assert isinstance(lst, list), 'Очікується list'
    assert len(lst) > 0,          'Список не може бути порожнім'
    return sum(lst) / len(lst)

# УВАГА: assert вимикається при запуску з флагом -O (optimize)
# Тому для продакшн-коду використовуйте raise, а не assert
# assert підходить для налагодження та тестів
```

Власні класи помилок успадковуються від Exception (або його підкласів). Вони дозволяють точно описати помилку, що є специфічною для вашої програми, та несуть додаткову інформацію.

```python
# Базова кастомна помилка
class AppError(Exception):
    '''Базовий клас для всіх помилок застосунку.'''
    pass

# Спеціалізовані помилки
class ValidationError(AppError):
    '''Помилка валідації вхідних даних.'''
    def __init__(self, field: str, message: str):
        self.field   = field
        self.message = message
        super().__init__(f'Поле "{field}": {message}')

class DatabaseError(AppError):
    '''Помилка бази даних.'''
    def __init__(self, query: str, reason: str):
        self.query  = query
        self.reason = reason
        super().__init__(f'Запит "{query}" failed: {reason}')

class InsufficientFundsError(AppError):
    '''Недостатньо коштів на рахунку.'''
    def __init__(self, balance: float, amount: float):
        self.balance = balance
        self.amount  = amount
        self.deficit = amount - balance
        super().__init__(
            f'Недостатньо коштів: баланс {balance:.2f}, '
            f'потрібно {amount:.2f} (бракує {self.deficit:.2f})'
        )

# Використання
def register_user(email: str, age: int):
    if '@' not in email:
        raise ValidationError('email', 'Некоректний формат')
    if age < 18:
        raise ValidationError('age', 'Потрібно бути повнолітнім')

def withdraw(balance: float, amount: float) -> float:
    if amount > balance:
        raise InsufficientFundsError(balance, amount)
    return balance - amount

# Перехоплення з доступом до атрибутів
try:
    withdraw(100.0, 250.0)
except InsufficientFundsError as e:
    print(f'Помилка: {e}')
    print(f'Бракує: {e.deficit:.2f} грн')

# Ієрархія: ловимо батьківський клас
try:
    register_user('bad-email', 16)
except ValidationError as e:
    print(f'Поле [{e.field}]: {e.message}')
```

```
except AppError as e:
    print(f'Загальна помилка застосунку: {e}')
```

Модуль `logging` – стандартний інструмент Python для запису подій (логів). Він є значно кращою альтернативою `print()` для реальних програм: дозволяє контролювати рівень деталізації, формат і місце запису без зміни коду.

Рівень	Значення	Коли використовувати
DEBUG	10	Деталі для розробки: значення змінних, кроки алгоритму
INFO	20	Нормальні події: запуск, завершення, важливі кроки
WARNING	30	Щось несподіване, але програма продовжує роботу
ERROR	40	Серйозна помилка, частина функціональності не працює
CRITICAL	50	Критична помилка, програма не може продовжити роботу

```
import logging

# — Базове налаштування —
logging.basicConfig(
    level=logging.DEBUG,          # мінімальний рівень виведення
    format='%(asctime)s [%(levelname)s] %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S',
)

# — Виведення повідомлень різних рівнів —
logging.debug('Запускаємо обробку даних...')
logging.info('Підключення до бази даних встановлено')
logging.warning('Файл конфігурації не знайдено, використовуємо defaults')
logging.error('Не вдалося відкрити файл: data.csv')
logging.critical('База даних недоступна! Зупиняємо сервер.')

# Результат у консолі:
# 2024-09-01 12:00:00 [DEBUG]    Запускаємо обробку даних...
# 2024-09-01 12:00:00 [INFO]     Підключення до бази даних встановлено
# 2024-09-01 12:00:00 [WARNING]  Файл конфігурації не знайдено...
# 2024-09-01 12:00:00 [ERROR]    Не вдалося відкрити файл: data.csv
# 2024-09-01 12:00:00 [CRITICAL] База даних недоступна!...

# — Логування винятків (зі stack trace!) —
try:
    result = 10 / 0
except ZeroDivisionError:
    logging.error('Помилка обчислення', exc_info=True)
    # або коротше:
    logging.exception('Помилка обчислення')  # автоматично exc_info=True
```

Реальні програми зазвичай записують логи одночасно у кілька місць: консоль та файл. Це досягається через handlers (обробники).

```python
import logging
from logging.handlers import RotatingFileHandler

# — Налаштування через getLogger (рекомендований спосіб) —
logger = logging.getLogger('my_app')  # іменований логер
logger.setLevel(logging.DEBUG)

# — Форматер —
formatter = logging.Formatter(
    fmt='%(asctime)s | %(name)s | %(levelname)-8s | %(message)s',
    datefmt='%d.%m.%Y %H:%M:%S'
)

# — Handler 1: виведення у консоль (INFO і вище) —
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.INFO)
console_handler.setFormatter(formatter)

# — Handler 2: запис у файл (DEBUG і вище) —
file_handler = logging.FileHandler('app.log', encoding='utf-8')
file_handler.setLevel(logging.DEBUG)
file_handler.setFormatter(formatter)

# — Handler 3: ротаційний файл (нові файли при переповненні) —
rotating = RotatingFileHandler(
    'app_rotating.log',
    maxBytes=1024 * 1024,   # 1 МБ
    backupCount=3,          # зберігати 3 старі файли
    encoding='utf-8'
)
rotating.setLevel(logging.WARNING)
rotating.setFormatter(formatter)

# — Додаємо handlers до логера —
logger.addHandler(console_handler)
logger.addHandler(file_handler)
logger.addHandler(rotating)

# — Використання —
logger.debug('Читаємо конфігурацію...')     # тільки у файл
logger.info('Сервер запущено на порту 8000') # консоль + файл
logger.warning('Вичерпано 80% пам\'яті')   # всі handlers
logger.error('HTTP 500: Internal Server Error')

# — Передача контексту у повідомлення —
user_id = 42
logger.info('Користувач %s виконав вхід', user_id)  # lazy formatting
logger.error('Помилка для user_id=%d: %s', user_id, 'timeout')
```

Правильна обробка помилок є ознакою зрілого коду. Ось ключові принципи:

```
# ── ПОГАНО vs ДОБРЕ ──

# ПОГАНО: ловити все без обробки (тихе проковтування помилки)
try:
    do_something()
except:
    pass            # помилка зникає безслідно!

# ДОБРЕ: ловити конкретний тип і реагувати
try:
    do_something()
except ValueError as e:
    logger.error('Неправильне значення: %s', e)
    return None

# ── ПОГАНО: обробляти те, що не очікується ──
try:
    x = 1 + 1       # це НІКОЛИ не кине винятok!
    y = x * 2
    result = x / y
except ZeroDivisionError:
    pass

# ДОБРЕ: мінімальний блок try
try:
    result = x / y  # тільки ризикований рядок
except ZeroDivisionError:
    result = 0

# ── Патерн: конвертація з дефолтним значенням ──
def safe_int(value, default=0):
    try:
        return int(value)
    except (ValueError, TypeError):
        return default

print(safe_int('42'))    # 42
print(safe_int('abc'))   # 0
print(safe_int(None, -1))  # -1

# ── Патерн: EAFP vs LBYL ──

# LBYL (Look Before You Leap) – перевіряємо перед дією
if 'key' in data:
    value = data['key']

# EAFP (Easier to Ask Forgiveness than Permission) – Pythonic стиль
try:
    value = data['key']
except KeyError:
    value = None

# Або ще коротше:
value = data.get('key')   # повертає None якщо ключ відсутній
```

Чек-ліст якісної обробки помилок:
Ловіть КОНКРЕТНИЙ тип, а не Exception або голий except.
Мінімізуйте блок try – лише рядки, що справді ризиковані.
Не проковтуйте помилки тихо – логуйте або повідомляйте.
Використовуйте finally або with для звільнення ресурсів.
Створюйте кастомні помилки для логіки вашого застосунку.
Логуйте exc_info=True для помилок – зберігайте traceback.

Завдання на лабораторну роботу

5.1. Реалізуйте функцію `safe_calculator()`, яка запитує у користувача два числа і символ операції (+, -, *, /, //, %). Обробіть всі можливі помилки:
- `ValueError` – якщо введено не число (перетворення через int або float);
- `ZeroDivisionError` – для / та //;
- невідомий символ операції – через `raise ValueError` з власним повідомленням.

Програма працює у циклі до введення 'q'. При кожній помилці виводить зрозуміле повідомлення і продовжує роботу. Після завершення виводить кількість успішних обчислень та кількість помилок.

5.2. Напишіть функцію `read_numbers_from_file(filename)`:
- `try`: відкрити файл та прочитати числа (по одному на рядку);
- `except FileNotFoundError`: вивести повідомлення і повернути порожній список;
- `except ValueError as e`: вивести який рядок файлу не вдалося перетворити у число;
- `else`: вивести 'Файл прочитано успішно: N чисел';
- `finally`: вивести 'Файл закрито' (навіть якщо виникла помилка).

Створіть тестовий файл numbers.txt із 5 числами (і навмисно одним нечисловим рядком). Виконайте функцію для цього файлу та для неіснуючого файлу.

5.3 Напишіть програму, що демонструє ієрархію винятків та базове логування:
- Налаштуйте `basicConfig`: рівень `DEBUG`, формат з часом та рівнем, запис у файл `errors.log`
- Напишіть функцію `demonstrate_exceptions()` що послідовно викликає: `ZeroDivisionError, ValueError, IndexError, KeyError, FileNotFoundError` – кожен у своєму `try-except` з `logging.error()`
- Покажіть що `except ArithmeticError` перехоплює `ZeroDivisionError`
- Покажіть що `except Exception` перехоплює всі програмні помилки крім `KeyboardInterrupt`
- Після виконання відкрийте `errors.log` і виведіть його вміст у консоль

Контрольні запитання

1. Яка різниця між синтаксичною помилкою та винятком? Наведіть приклад кожного.
2. Поясніть роль кожного блоку конструкції `try-except-else-finally`. Чи обов'язкові всі чотири?
3. Що таке 'проковтування помилки' (`swallowing exception`)? Чому це є поганою практикою?
4. Що таке ієрархія винятків? Чому `catch except Exception` не перехоплює `KeyboardInterrupt`?
5. Для чого використовується оператор `raise`? Чим відрізняється `raise ValueError(...)` від простого `raise` всередині `except`?
6. Для чого потрібні кастомні класи винятків? Яку перевагу дає зберігання додаткових даних у кастомному винятку?
7. Що таке `assert`? Чим він відрізняється від `raise`? Чому не слід покладатися на `assert` у продакшн-коді?
8. Які п'ять рівнів логування є у модулі `logging`? В якому порядку вони зростають за серйозністю?
9. Що таке `handler` у `logging`? Яка різниця між `StreamHandler` та `FileHandler`? Як записувати логи одночасно у консоль і файл?
10. Чим `logging.exception()` відрізняється від `logging.error()`? Що таке `exc_info=True` і навіщо це потрібно?

Лабораторна робота №6.

Робота з файлами

Мета роботи: Навчитися відкривати, читати та записувати текстові і бінарні файли у Python, зрозуміти різницю між режимами відкриття файлів та освоїти менеджер контексту with як основний безпечний спосіб роботи з файлами. Ознайомитися з модулями стандартної бібліотеки `os`, `os.path`, `pathlib` та `shutil` для безпечного створення, копіювання, переміщення, перейменування файлів та директорій, а також навчитися обходити файлову систему.Засвоїти роботу з різними кодуваннями тексту, навчитися виконувати пошук рядків у файлах та застосовувати регулярні вирази (модуль re) для розбору і фільтрації текстових даних у файлах.

Теоретичні відомості:

Вбудована функція `open()` відкриває файл і повертає файловий об'єкт. Режим відкриття визначає, що дозволено робити з файлом: читати, писати або дописувати.

Синтаксис / Режим	Опис / Дія
`'r'`	Читання (read). Файл повинен існувати. Режим за замовчуванням.
`'w'`	Запис (write). Якщо файл існує – очищає його. Якщо ні – створює.
`'a'`	Дописування (append). Дані додаються в кінець. Файл створюється якщо відсутній.
`'x'`	Ексклюзивне створення. Кидає FileExistsError якщо файл вже є.
`'r+'`	Читання і запис (файл повинен існувати).
`'w+'`	Читання і запис (файл очищається або створюється).
`'b'`	Бінарний режим (додається до інших): 'rb', 'wb', 'ab'.
`'t'`	Текстовий режим (за замовчуванням, можна не вказувати).

```
# Базовий синтаксис open()
# open(file, mode='r', encoding=None, errors='strict')
# Читання текстового файлу
f = open('data.txt', 'r', encoding='utf-8')
content = f.read()
f.close()              # ОБОВ'ЯЗКОВО закривати вручну
# Запис у файл
f = open('output.txt', 'w', encoding='utf-8')
f.write('Привіт, файл!\n')
f.close()

# Проблема: якщо між open() і close() виникне виняток –
# файл не закриється, ресурс «протече».
# Рішення: менеджер контексту with (розглянуто далі).
```

Менеджер контексту `with` гарантує, що файл буде закрито автоматично – навіть якщо в тілі блоку виникне виняток. Це найбезпечніший і рекомендований спосіб роботи з файлами в Python.

```python
# with автоматично викликає f.close() при виході з блоку
with open('notes.txt', 'w', encoding='utf-8') as f:
    f.write('Перший рядок\n')
    f.write('Другий рядок\n')
# Тут файл вже закрито – навіть якщо виникне виняток

# Читання файлу
with open('notes.txt', 'r', encoding='utf-8') as f:
    content = f.read()
print(content)

# Два файли в одному with (Python 3.10+: можна у дужках)
with (open('source.txt', 'r', encoding='utf-8') as src,
      open('copy.txt',   'w', encoding='utf-8') as dst):
    dst.write(src.read())

# Перевірка: після with файл закрито
with open('notes.txt') as f:
    pass
print(f.closed)    # True
```

Правило!

Завжди використовуйте `with open(...) as f` *замість* `open()` + `f.close()`
Це захищає від витоку файлових дескрипторів та пошкодження даних.
Ніколи не забувайте `encoding='utf-8'` *для текстових файлів з кирилицею.*

Методи читання файлів

Синтаксис / Режим	Опис / Дія
`f.read()`	Читає весь файл як один рядок (str). *Обережно з великими файлами!*
`f.read(n)`	Читає n символів (або байт у бінарному режимі).
`f.readline()`	Читає один рядок (включно з `'\n'` у кінці).
`f.readlines()`	Повертає список усіх рядків файлу.
`for ln in f:`	Ітерація по рядках – найефективніший спосіб для великих файлів.
`f.tell()`	Повертає поточну позицію (в байтах) у файлі.
`f.seek(n)`	Переміщує курсор на позицію n байт від початку.

```python
# 1. read() – весь файл як рядок
with open('poem.txt', 'r', encoding='utf-8') as f:
    text = f.read()
print(f'Символів: {len(text)}')

# 2. readline() – рядок за рядком вручну
with open('poem.txt', 'r', encoding='utf-8') as f:
    first  = f.readline()   # перший рядок
    second = f.readline()   # другий рядок
    print(first.strip(), second.strip())

# 3. readlines() – список рядків
with open('poem.txt', 'r', encoding='utf-8') as f:
    lines = f.readlines()   # ['рядок1\n', 'рядок2\n', ...]
print(f'Рядків: {len(lines)}')
print(lines[0].strip())    # перший рядок без \n
# 4. Ітерація – НАЙКРАЩИЙ спосіб для великих файлів
# Зчитує по одному рядку, не завантажуючи весь файл у пам'ять
```

```python
with open('poem.txt', 'r', encoding='utf-8') as f:
    for i, line in enumerate(f, start=1):
        print(f'{i:3}: {line}', end='')

# 5. seek() і tell()
with open('poem.txt', 'r', encoding='utf-8') as f:
    print(f.read(10))    # читаємо 10 символів
    print(f.tell())      # поточна позиція
    f.seek(0)            # повертаємось на початок
    print(f.read(10))    # читаємо ті ж 10 символів знову
```

Методи запису. Дописування у файл

```python
# write() – записує рядок (без автоматичного \n!)
with open('log.txt', 'w', encoding='utf-8') as f:
    f.write('Рядок перший\n')
    f.write('Рядок другий\n')
    chars = f.write('Рядок третій\n')
    print(f'Записано символів: {chars}')  # write() повертає кількість

# writelines() – записує список рядків
lines = ['Яблуко\n', 'Груша\n', 'Банан\n']
with open('fruits.txt', 'w', encoding='utf-8') as f:
    f.writelines(lines)    # \n НЕ додається автоматично!

# Режим 'a' – ДОПИСУВАННЯ (append), не затирає файл
import datetime
now = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')

with open('events.log', 'a', encoding='utf-8') as f:
    f.write(f'[{now}] Програма запущена\n')

# print() теж вміє писати у файл через параметр file=
with open('report.txt', 'w', encoding='utf-8') as f:
    print('Звіт', file=f)
    print('=' * 30, file=f)
    for i in range(1, 4):
        print(f'Запис {i}: {i**2}', file=f)
```

Кодування визначає, як символи Unicode перетворюються на байти при збереженні. Без явного зазначення encoding Python використовує системне кодування (у Windows це часто `cp1251`, у Linux/Mac – `utf-8`).

Синтаксис / Режим	Опис / Дія
`'utf-8'`	Універсальне. Підтримує всі мови світу. Рекомендовано завжди.
`'utf-8-sig'`	UTF-8 з BOM-маркером. Потрібен для Excel та деяких Windows-програм.
`'cp1251'`	Кирилиця Windows (Windows-1251). Часто у старих файлах.
`'latin-1'`	ISO 8859-1. Зустрічається у файлах із Заходу.
`errors='strict'`	За замовчуванням: кидає UnicodeDecodeError при помилці.
`errors='ignore'`	Ігнорує символи, які не вдається декодувати.
`errors='replace'`	Замінює проблемні символи на '?' або ''.

```python
# Запис у різних кодуваннях
text = 'Привіт, це текст українською!'

# UTF-8 – рекомендовано
with open('utf8.txt', 'w', encoding='utf-8') as f:
    f.write(text)

# UTF-8 з BOM (для сумісності з Excel)
with open('utf8_bom.txt', 'w', encoding='utf-8-sig') as f:
    f.write(text)

# CP1251 – Windows-кирилиця
with open('cp1251.txt', 'w', encoding='cp1251') as f:
    f.write(text)

# Визначення кодування невідомого файлу (бібліотека chardet)
# pip install chardet
# import chardet
# with open('unknown.txt', 'rb') as f:
#     raw = f.read()
# detected = chardet.detect(raw)
# print(detected)   # {'encoding': 'UTF-8', 'confidence': 0.99}

# Безпечне читання з errors='replace'
with open('cp1251.txt', 'r', encoding='utf-8', errors='replace') as f:
    content = f.read()    # неправильні байти → '\ufffd'
print(content)

# Правильне читання cp1251-файлу
with open('cp1251.txt', 'r', encoding='cp1251') as f:
    content = f.read()    # тепер текст коректний
print(content)
```

Модуль `os` надає функції для взаємодії з операційною системою: керування файлами, директоріями та середовищем виконання. `os.path` – підмодуль для роботи зі шляхами файлів.

```python
import os

# ── Інформація про поточне оточення ──
print(os.getcwd())              # поточна директорія
print(os.listdir('.'))          # список файлів у поточній директорії
print(os.listdir('C:/Users'))   # список файлів у вказаній папці

# ── Робота з директоріями ──
os.mkdir('new_folder')          # створити папку
os.makedirs('a/b/c', exist_ok=True)  # створити вкладені папки
os.chdir('new_folder')          # перейти у папку
os.chdir('..')                  # повернутися на рівень вгору
os.rmdir('new_folder')          # видалити ПОРОЖНЮ папку

# ── Операції з файлами ──
os.rename('old.txt', 'new.txt') # перейменувати файл або папку
os.remove('temp.txt')           # видалити файл

# ── os.path: робота зі шляхами ──
path = '/home/user/projects/lab6/data.txt'
```

```
print(os.path.basename(path))   # data.txt
print(os.path.dirname(path))    # /home/user/projects/lab6
print(os.path.split(path))
# ('/home/user/projects/lab6', 'data.txt')

print(os.path.splitext('data.txt'))  # ('data', '.txt')
print(os.path.join('lab6', 'data', 'file.txt'))
# lab6/data/file.txt  (або lab6\data\file.txt на Windows)

print(os.path.exists('data.txt'))    # True/False
print(os.path.isfile('data.txt'))    # чи є файлом
print(os.path.isdir('lab6'))         # чи є директорією
print(os.path.getsize('data.txt'))   # розмір у байтах
print(os.path.abspath('data.txt'))   # абсолютний шлях
```

Модуль `pathlib` пропонує об'єктно-орієнтований інтерфейс до файлової системи. Path-об'єкти – більш зручна та читабельна альтернатива `os.path`.

```
from pathlib import Path

# Створення Path-об'єкта
p = Path('.')                    # поточна директорія
p = Path('/home/user/projects')  # абсолютний шлях
p = Path.home()                  # домашня директорія користувача
p = Path.cwd()                   # поточна робоча директорія

# Оператор / для з'єднання частин шляху
project = Path('lab6')
data_file = project / 'data' / 'students.txt'
print(data_file)    # lab6/data/students.txt

# Властивості Path-об'єкта
p = Path('lab6/data/report.csv')
print(p.name)        # report.csv
print(p.stem)        # report
print(p.suffix)      # .csv
print(p.parent)      # lab6/data
print(p.parents[0])  # lab6/data
print(p.parents[1])  # lab6
print(p.is_absolute()) # False

# Перевірки та інформація
p = Path('data.txt')
print(p.exists())    # True / False
print(p.is_file())   # чи є файлом
print(p.is_dir())    # чи є директорією
print(p.stat().st_size)  # розмір у байтах

# Читання та запис через pathlib
p = Path('hello.txt')
p.write_text('Привіт, pathlib!', encoding='utf-8')
print(p.read_text(encoding='utf-8'))  # Привіт, pathlib!

# Перебір файлів у директорії
for f in Path('.').iterdir():
    if f.is_file():
        print(f.name, f.stat().st_size, 'bytes')

# glob() - пошук файлів за шаблоном
```

```python
for txt in Path('.').glob('*.txt'):
    print(txt)

# rglob() – рекурсивний пошук у підпапках
for py in Path('.').rglob('*.py'):
    print(py)

# Створення директорій
Path('output/logs').mkdir(parents=True, exist_ok=True)
```

Модуль `shutil` (shell utilities) надає функції вищого рівня для операцій над файлами та директоріями. На відміну від `os`, він вміє рекурсивно копіювати та видаляти директорії.

Синтаксис / Режим	Опис / Дія
`shutil.copy(src, dst)`	Копіює файл src у dst (без метаданих).
`shutil.copy2(src, dst)`	Копіює файл зі збереженням метаданих (дата, атрибути).
`shutil.copytree(src, dst)`	Рекурсивно копіює директорію з усім вмістом.
`shutil.move(src, dst)`	Переміщує або перейменовує файл/директорію.
`shutil.rmtree(path)`	Рекурсивно видаляє директорію з усім вмістом.
`shutil.make_archive(...)`	Створює архів (zip, tar, gztar).
`shutil.unpack_archive(...)`	Розпаковує архів у вказану папку.
`shutil.disk_usage(path)`	Повертає загальний, використаний та вільний простір.

```python
import shutil
from pathlib import Path

# ── Копіювання файлів ──
shutil.copy('report.txt', 'backup/report.txt')   # копія
shutil.copy2('data.csv', 'archive/')             # зберігає метадані

# ── Безпечне копіювання з перевіркою ──
src = Path('data.txt')
dst = Path('backup/data.txt')

dst.parent.mkdir(parents=True, exist_ok=True)   # створимо папку якщо немає
if src.exists():
    shutil.copy2(src, dst)
    print(f'Скопійовано: {src} → {dst}')
else:
    print(f'Файл не знайдено: {src}')

# ── Копіювання директорії ──
shutil.copytree('project/', 'project_backup/',
                ignore=shutil.ignore_patterns('*.pyc', '__pycache__'))

# ── Переміщення і перейменування ──
shutil.move('old_name.txt', 'new_name.txt')     # перейменування
shutil.move('file.txt', 'archive/')             # переміщення у папку

# ── Видалення директорії рекурсивно ──
# УВАГА: операція незворотна!
tmp = Path('temp_folder')
if tmp.exists():
    shutil.rmtree(tmp)
    print('Тимчасову папку видалено')
```

```
# ── Архівування ──
shutil.make_archive('project_v1', 'zip', 'project/')   # → project_v1.zip
shutil.make_archive('backup', 'gztar', 'data/')        # → backup.tar.gz

# ── Розпакування ──
shutil.unpack_archive('project_v1.zip', 'restored/')

# ── Інформація про диск ──
usage = shutil.disk_usage('/')
print(f'Всього:    {usage.total / 1024**3:.1f} ГБ')
print(f'Вільно:    {usage.free  / 1024**3:.1f} ГБ')
print(f'Зайнято:   {usage.used  / 1024**3:.1f} ГБ')
```

Базовий пошук рядка у файлі можна виконати простою перевіркою `in`, але для складніших шаблонів використовують регулярні вирази (модуль re).

Простий пошук рядків

```
# ── Пошук рядків, що містять ключове слово ──
def find_lines(filename, keyword, encoding='utf-8'):
    '''Повертає список рядків, що містять keyword'''
    results = []
    with open(filename, 'r', encoding=encoding) as f:
        for lineno, line in enumerate(f, start=1):
            if keyword.lower() in line.lower():  # без урахування регістру
                results.append((lineno, line.rstrip()))
    return results

hits = find_lines('notes.txt', 'python')
for lineno, line in hits:
    print(f'Рядок {lineno:3}: {line}')

# ── Підрахунок слів у файлі ──
def word_count(filename, encoding='utf-8'):
    with open(filename, 'r', encoding=encoding) as f:
        text = f.read()
    words = text.split()
    unique = set(w.lower().strip('.,!?;:') for w in words)
    return {'total': len(words), 'unique': len(unique),
            'lines': text.count('\n') + 1,
            'chars': len(text)}

stats = word_count('essay.txt')
print(f'Слів всього:  {stats["total"]}')
print(f'Унікальних:   {stats["unique"]}')
print(f'Рядків:       {stats["lines"]}')
```

Регулярний вираз (`regex`) – це шаблон для пошуку та заміни тексту. Модуль re забезпечує потужний інструментарій для роботи з такими шаблонами.

Синтаксис / Режим	Опис / Дія
`re.search(pat, s)`	Перше входження pat у рядку s. Повертає Match або None.
`re.match(pat, s)`	Перевіряє pat лише на ПОЧАТКУ рядка.
`re.findall(pat, s)`	Список УСІХ входжень (рядки).
`re.finditer(pat, s)`	Ітератор Match-об'єктів усіх входжень.
`re.sub(pat, repl, s)`	Замінює всі входження pat на repl у рядку s.

`re.compile(pat)`	Компілює шаблон для багаторазового використання.
`re.ignorecase`	Прапор: пошук без урахування регістру (re.I).
`re.multiline`	Прапор: ^ та $ відповідають початку/кінцю кожного рядка.

Основні елементи шаблонів

```
#  .       – будь-який символ (крім \n)
#  \d      – цифра [0-9]
#  \D      – не цифра
#  \w      – літера, цифра або _  [a-zA-Z0-9_]
#  \s      – пробільний символ (пробіл, \t, \n)
#  ^       – початок рядка
#  $       – кінець рядка
#  *       – 0 або більше повторень
#  +       – 1 або більше повторень
#  ?       – 0 або 1 повторення
#  {n,m}   – від n до m повторень
#  [abc]   – один із символів a, b, c
#  [^abc]  – будь-який символ, крім a, b, c
#  (...)   – група (можна витягнути через .group())
#  |       – або (a|b – a або b)
```

Практичні приклади пошуку у файлах

```
import re
# ── Витягти всі email-адреси з файлу ──
email_pattern = re.compile(
    r'[a-zA-Z0-9._%+\-]+@[a-zA-Z0-9.\-]+\.[a-zA-Z]{2,}'
)

def extract_emails(filename):
    with open(filename, 'r', encoding='utf-8') as f:
        text = f.read()
    return email_pattern.findall(text)

emails = extract_emails('contacts.txt')
print(f'Знайдено email: {len(emails)}')
for e in emails:
    print(f'  {e}')
# ── Витягти всі номери телефонів ──
phone_pattern = re.compile(
    r'(?:\+38)?[\s\-]?\(?0\d{2}\)?[\s\-]?\d{3}[\s\-]?\d{2}[\s\-]?\d{2}'
)

def extract_phones(filename):
    with open(filename, 'r', encoding='utf-8') as f:
        text = f.read()
    return phone_pattern.findall(text)

# ── Знайти рядки, що відповідають шаблону ──
def grep(filename, pattern, flags=re.IGNORECASE):
    '''Аналог unix grep – знаходить рядки за regex-шаблоном'''
    pat = re.compile(pattern, flags)
    results = []
    with open(filename, 'r', encoding='utf-8') as f:
        for no, line in enumerate(f, 1):
            if pat.search(line):
                results.append((no, line.rstrip()))
    return results
```

```
# Знайти рядки, що містять дати формату DD.MM.YYYY
date_hits = grep('report.txt', r'\d{2}\.\d{2}\.\d{4}')
for no, line in date_hits:
    print(f'{no}: {line}')
# ── Заміна шаблонів у файлі ──
def replace_in_file(filename, pattern, replacement, out_file=None):
    '''Замінює всі входження pattern у файлі'''
    with open(filename, 'r', encoding='utf-8') as f:
        text = f.read()

    new_text, count = re.subn(pattern, replacement, text)
    print(f'Замінено входжень: {count}')
    target = out_file or filename
    with open(target, 'w', encoding='utf-8') as f:
        f.write(new_text)

# Замінити всі дати з DD.MM.YYYY на YYYY-MM-DD
replace_in_file(
    'report.txt',
    r'(\d{2})\.(\d{2})\.(\d{4})',   # групи: день, місяць, рік
    r'\3-\2-\1',                    # переставляємо групи
    out_file='report_fixed.txt'
)
# ── Парсинг CSV вручну через re ──
log_pattern = re.compile(
    r'^(\d{4}-\d{2}-\d{2})\s+(\w+)\s+(.+)$'
)

with open('app.log', 'r', encoding='utf-8') as f:
    for line in f:
        m = log_pattern.match(line)
        if m:
            date, level, msg = m.groups()
            if level == 'ERROR':
                print(f'[{date}] ПОМИЛКА: {msg}')
```

При роботі з файлами можуть виникати різні помилки. Їх необхідно обробляти, щоб програма не падала з некоректним повідомленням.

Синтаксис / Режим	Опис / Дія
`FileNotFoundError`	Файл або директорія не існує.
`PermissionError`	Немає прав доступу на читання або запис.
`IsADirectoryError`	Спроба відкрити директорію як файл.
`FileExistsError`	Файл вже існує (режим 'x').
`UnicodeDecodeError`	Файл не вдається декодувати в заданому кодуванні.
`OSError`	Базовий клас для всіх файлових помилок (ловить усі вище).

```
from pathlib import Path

def safe_read(filepath, encoding='utf-8'):
    '''Безпечне читання файлу з обробкою помилок'''
    path = Path(filepath)
    try:
        with open(path, 'r', encoding=encoding) as f:
            return f.read()
```

```
    except FileNotFoundError:
        print(f'Файл не знайдено: {path}')
    except PermissionError:
        print(f'Немає прав доступу: {path}')
    except UnicodeDecodeError:
        print(f'Помилка кодування. Спробуйте encoding="cp1251"')
    except OSError as e:
        print(f'Помилка файлової системи: {e}')
    return None

def safe_write(filepath, content, encoding='utf-8'):
    '''Безпечний запис з резервною копією'''
    path = Path(filepath)
    backup = path.with_suffix('.bak')
    try:
        # Зробити резервну копію якщо файл існує
        if path.exists():
            import shutil
            shutil.copy2(path, backup)
        with open(path, 'w', encoding=encoding) as f:
            f.write(content)
        print(f'Записано успішно: {path}')
        return True
    except PermissionError:
        print(f'Немає прав запису: {path}')
    except OSError as e:
        print(f'Помилка запису: {e}')
        # Відновити з резервної копії
        if backup.exists():
            shutil.copy2(backup, path)
    return False

# Використання
content = safe_read('data.txt')
if content is not None:
    processed = content.upper()
    safe_write('data_upper.txt', processed)
```

Завдання на лабораторну роботу

Перед виконанням завдань створіть у своєму проекті папку lab6_data/ і підготуйте в ній текстові файли з тестовим вмістом (студенти, оцінки, довільний текст). Усі завдання виконуйте з цими файлами.

6.1. Читання і запис текстового файлу. Виконайте такі дії:

- вручну (не кодом!) створіть файл `students.txt` із 8–10 рядків у форматі: `"Прізвище Ім'я, оцінка"` (оцінка від 50 до 100);
- напишіть програму, яка читає файл рядок за рядком і виводить кожен рядок з номером;
- підрахуйте і виведіть кількість рядків, кількість слів (загалом у файлі), розмір файлу в байтах (через `os.path.getsize()`);
- запишіть у новий файл `students_upper.txt` вміст `students.txt` у верхньому регістрі – ОБОВ'ЯЗКОВО через `with` і обробку `UnicodeDecodeError`.

6.2. Ведення журналу подій. Реалізуйте систему логування:
- Напишіть функцію `log_event(filename, level, message)`, яка дописує у файл рядок формату: `[YYYY-MM-DD HH:MM:SS] [LEVEL] message`;
- використовуйте режим `'a'` і модуль `datetime` для отримання поточного часу;
- викличте функцію мінімум 10 разів з різними рівнями: `INFO`, `WARNING`, `ERROR`;
- напишіть функцію `read_log(filename, level=None)`, яка читає лог і повертає список записів – усіх (якщо `level=None`) або тільки заданого рівня;
- виведіть кількість записів кожного рівня.

6.3. Операції з файловою системою (`os` та `pathlib`). Напишіть програму, яка:
- створює структуру директорій: `lab6_project/src/`, `lab6_project/data/`, `lab6_project/output/` (через `pathlib.mkdir` з `exist_ok=True`);
- виводить дерево цієї структури (назву кожного елемента з відступом залежно від глибини);
- для кожного файлу у `lab6_data/` виводить: ім'я файлу, розширення, розмір у байтах і дату останньої зміни (через `Path.stat().st_mtime` та `datetime.fromtimestamp()`);
- використайте `glob('*.txt')` для пошуку тільки текстових файлів.

6.4. Безпечне копіювання та резервне збереження (`shutil`). Реалізуйте утиліту резервного копіювання:
- функція `backup_file(src_path)` – копіює файл у папку backup/ з додаванням часової мітки до імені: `file.txt → backup/file_2026-11-15_14-30.txt`;
- функція `backup_folder(src_dir, dst_dir)` – копіює директорію через `shutil.copytree`, ігноруючи `*.tmp` і `*.log` файли;
- функція `restore_latest(filename)` – знаходить найсвіжішу резервну копію (через `glob` і `sorted`) і відновлює її;
- Продемонструйте роботу всіх трьох функцій і обробіть можливі виняткові ситуації (FileNotFoundError, PermissionError)

6.5. Пошук і аналіз тексту за допомогою регулярних виразів. Створіть файл `contacts.txt` із 10–15 рядками у довільному форматі, де зустрічаються: email-адреси, номери телефонів (у різних форматах), дати (DD.MM.YYYY).
Напишіть програму, яка:
- витягує всі email-адреси з файлу (шаблон: `user@domain.tld`);
- витягує всі номери телефонів (шаблон враховує різні формати: +38(067)..., 0671234567, 067-123-45-67);
- знаходить всі дати у форматі DD.MM.YYYY і конвертує їх у ISO-формат YYYY-MM-DD через `re.sub()` із групами;
- Записує структурований результат у `results.txt`: три секції зі знайденими email, телефонами і датами.

Контрольні запитання

1. Що таке менеджер контексту `with`? Чому він є кращою альтернативою до явного виклику `f.close()`?
2. Назвіть і поясніть 5 режимів відкриття файлів. Чим відрізняється режим `'w'` від `'a'`? Що станеться з існуючим файлом при відкритті в режимі `'w'`?
3. Яка різниця між `f.read()`, `f.readline()` і `f.readlines()`? Коли доцільно використовувати ітерацію по файловому об'єкту `for line in f`?
4. Що таке кодування тексту? Чому важливо явно вказувати `encoding='utf-8'` при відкритті файлів? Що станеться якщо цього не зробити на Windows?
5. Чим `pathlib.Path` відрізняється від `os.path`? Яке переваги дає оператор `/` для Path-об'єктів?
6. Назвіть три функції модуля `shutil` для роботи з файлами. Чим `shutil.copy()` відрізняється від `shutil.copy2()`?
7. Яка різниця між `os.remove()` та `shutil.rmtree()`? В яких ситуаціях потрібна кожна з них?
8. Що таке регулярний вираз? Поясніть значення символів \d, \w, \s, +, * та ? у шаблоні.
9. Яка різниця між `re.search()` та `re.match()`? Коли використовувати `re.findall()`, а коли `re.finditer()`?
10. Назвіть три винятки, які можуть виникнути при роботі з файлами. Як правильно їх обробляти за допомогою `try-except`?

Лабораторна робота №7.

Основи об'єктно-орієнтованого програмування в Python

Мета роботи: Засвоїти фундаментальні концепції об'єктно-орієнтованого програмування: навчитися визначати класи, створювати об'єкти та керувати їхніми атрибутами й методами; зрозуміти принципи інкапсуляції через механізми public, protected та private. Ознайомитися з різними видами методів (екземпляра, класовими, статичними), навчитися використовувати декоратор `@property` для контрольованого доступу до атрибутів; опанувати механізм успадкування (одиночне й множинне), порядок вирішення методів (MRO) та перевизначення методів. Вивчити абстрактні класи, спеціальні (dunder/magic) методи, протоколи та найкращі практики проектування класів у Python; навчитися застосовувати ООП для побудови реальних програм зі складною ієрархією об'єктів.

Теоретичні відомості:

Об'єктно-орієнтоване програмування (ООП) – парадигма, де програма моделюється як набір об'єктів, що взаємодіють між собою. Кожен об'єкт поєднує дані (атрибути) і поведінку (методи).

Чотири стовпи ООП:

- Інкапсуляція – приховання внутрішньої реалізації, доступ через інтерфейс
- Успадкування – клас-нащадок отримує властивості батька і може їх розширити
- Поліморфізм – різні класи можуть реалізовувати однаковий інтерфейс по-різному
- Абстракція – приховання деталей реалізації, робота з концепціями

```python
# ── Визначення класу ──
class Student:
    '''Клас для представлення студента.'''

    # Атрибут класу (спільний для всіх екземплярів)
    university = 'КПІ'
    _count = 0          # лічильник екземплярів

    # Конструктор: викликається при створенні об'єкта
    def __init__(self, name: str, age: int, gpa: float = 0.0):
        # Атрибути екземпляра (унікальні для кожного об'єкта)
        self.name = name          # public
        self.age  = age           # public
        self._gpa = gpa           # protected (конвенція)
        Student._count += 1

    # Метод екземпляра
    def greet(self) -> str:
        return f'Привіт! Я {self.name}, {self.age} років.'

    def __repr__(self) -> str:
        return f'Student(name={self.name!r}, age={self.age})'

# ── Створення об'єктів (екземплярів) ──
s1 = Student('Іван', 20, 3.8)
s2 = Student('Марія', 21, 4.5)

print(s1.greet())          # Привіт! Я Іван, 20 років.
```

```
print(s1.name)              # Іван
print(Student.university)   # КПІ  - атрибут класу
print(s1.university)        # КПІ  - доступний і через екземпляр
print(Student._count)       # 2    - скільки студентів створено
print(repr(s1))             # Student(name='Іван', age=20)

# Атрибути можна додавати динамічно (але краще уникати)
s1.email = 'ivan@kpi.ua'    # Нормально, але не рекомендується
print(s1.__dict__)          # {'name': 'Іван', 'age': 20, ...}
```

Інкапсуляція: public, protected, private.

Python не має справжніх модифікаторів доступу, як Java чи C++. Замість цього використовуються конвенції іменування та механізм name mangling.

Синтаксис / Концепція	Опис / Приклад
`name`	Public. Доступний звідусіль. Звичайний атрибут або метод.
`_name`	Protected. Конвенція: «тільки для внутрішнього використання та підкласів». Python не забороняє доступ ззовні, але це сигнал розробнику.
`__name`	Private (name mangling). Python перейменовує на _ClassName__name. Захищає від випадкового перевизначення у підкласах.
`__name__`	Dunder / magic method. Зарезервовано Python. Не використовуйте для своїх атрибутів.

```
class BankAccount:
    def __init__(self, owner: str, balance: float = 0.0):
        self.owner    = owner         # public: ім'я власника
        self._balance = balance       # protected: баланс
        self.__pin    = '0000'        # private: PIN-код (name mangling)
        self.__history = []           # private: історія операцій

    def deposit(self, amount: float) -> None:
        if amount <= 0:
            raise ValueError('Сума повинна бути додатною')
        self._balance += amount
        self.__history.append(f'+{amount:.2f}')

    def withdraw(self, amount: float, pin: str) -> bool:
        if pin != self.__pin:
            print('Невірний PIN!')
            return False
        if amount > self._balance:
            print('Недостатньо коштів!')
            return False
        self._balance -= amount
        self.__history.append(f'-{amount:.2f}')
        return True

    def get_history(self) -> list:
        return self.__history.copy()  # повертаємо КОПІЮ

    def __repr__(self) -> str:
        return f'BankAccount({self.owner!r}, {self._balance:.2f}₴)'

acc = BankAccount('Іван', 1000.0)
acc.deposit(500)
```

```python
acc.withdraw(200, '0000')

print(acc)                      # BankAccount('Іван', 1300.00₴)
print(acc.owner)                # Іван          - public OK
print(acc._balance)             # 1300.0        - protected (можна, але небажано)
# print(acc.__pin)              # AttributeError!
print(acc._BankAccount__pin)    # '0000' - name mangling (технічно можливо)
print(acc.get_history())        # ['+500.00', '-200.00']
```

Правило!
Використовуйте `_name` для атрибутів, що є «внутрішніми деталями реалізації».
Використовуйте `__name` лише коли треба захиститися від перевизначення у підкласах.
Для контрольованого доступу до `_атрибутів` використовуйте `@property` (розглянуто далі).

Спеціальні методи – методи Python з подвійним підкресленням на початку і кінці. Вони дозволяють класам підтримувати вбудовані операції: +, -, len(), str(), порівняння тощо.

Синтаксис / Концепція	Опис / Приклад
`__init__(self, ...)`	Конструктор. Ініціалізує новостворений об'єкт.
`__repr__(self)`	Рядкове представлення для розробника: repr(obj). Має бути однозначним.
`__str__(self)`	Рядкове представлення для користувача: str(obj), print(obj).
`__len__(self)`	Підтримка len(obj).
`__eq__(self, other)`	Оператор ==. Без нього порівнюються адреси об'єктів.
`__lt__, __le__, etc.`	Оператори <, <=, >, >= (less than, less or equal).
`__add__(self, other)`	Оператор +: obj1 + obj2.
`__getitem__(self, key)`	Підтримка obj[key].
`__contains__(self, x)`	Підтримка x in obj.
`__iter__(self)`	Підтримка ітерації: for x in obj.
`__enter__/__exit__`	Підтримка менеджера контексту with obj: ...
`__del__(self)`	Деструктор. Викликається перед знищенням об'єкта.

```python
from functools import total_ordering

@total_ordering   # автоматично генерує >, >=, <= з __eq__ і __lt__
class Vector2D:
    '''Двовимірний вектор з повним набором операцій.'''

    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y

    def __repr__(self) -> str:              # для розробника
        return f'Vector2D({self.x}, {self.y})'

    def __str__(self) -> str:               # для користувача
        return f'({self.x}, {self.y})'

    def __add__(self, other: 'Vector2D') -> 'Vector2D':
        return Vector2D(self.x + other.x, self.y + other.y)

    def __sub__(self, other: 'Vector2D') -> 'Vector2D':
        return Vector2D(self.x - other.x, self.y - other.y)
```

```
    def __mul__(self, scalar: float) -> 'Vector2D':  # v * k
        return Vector2D(self.x * scalar, self.y * scalar)

    def __rmul__(self, scalar: float) -> 'Vector2D': # k * v
        return self.__mul__(scalar)

    def __abs__(self) -> float:              # abs(v) → довжина
        return (self.x**2 + self.y**2) ** 0.5

    def __eq__(self, other) -> bool:
        return (self.x, self.y) == (other.x, other.y)

    def __lt__(self, other) -> bool:         # порівняння за довжиною
        return abs(self) < abs(other)

    def __bool__(self) -> bool:              # нульовий вектор → False
        return self.x != 0 or self.y != 0

    def dot(self, other: 'Vector2D') -> float:  # скалярний добуток
        return self.x * other.x + self.y * other.y

v1 = Vector2D(3, 4)
v2 = Vector2D(1, 2)

print(v1)               # (3, 4)
print(repr(v1))         # Vector2D(3, 4)
print(v1 + v2)          # (4, 6)
print(v1 - v2)          # (2, 2)
print(v1 * 2)           # (6, 8)
print(3 * v2)           # (3, 6)
print(abs(v1))          # 5.0
print(v1 > v2)          # True  (5.0 > 2.23)
print(v1 == Vector2D(3, 4))  # True
print(v1.dot(v2))       # 11.0
```

`@property` перетворює метод у «розумний атрибут» – при читанні атрибута викликається getter-метод, при записі – setter, при видаленні – deleter. Це ключовий інструмент інкапсуляції в Python.

Перевага перед прямим доступом до атрибута: можна додати валідацію або обчислення без зміни інтерфейсу класу.

```
class Temperature:
    '''Клас для роботи з температурою з автоматичним перетворенням.'''

    def __init__(self, celsius: float = 0.0):
        self._celsius = celsius    # зберігаємо у захищеному атрибуті

    # — getter: читання атрибута —
    @property
    def celsius(self) -> float:
        return self._celsius

    # — setter: запис зі валідацією —
    @celsius.setter
    def celsius(self, value: float) -> None:
        if value < -273.15:
            raise ValueError(f'Нижче абсолютного нуля: {value}°C')
```

```
        self._celsius = float(value)
    # ── Обчислюваний атрибут (тільки getter, без setter) ──
    @property
    def fahrenheit(self) -> float:
        return self._celsius * 9/5 + 32

    @property
    def kelvin(self) -> float:
        return self._celsius + 273.15

    # ── setter для fahrenheit ──
    @fahrenheit.setter
    def fahrenheit(self, value: float) -> None:
        self.celsius = (value - 32) * 5/9   # через setter celsius (з валідацією!)

    def __repr__(self) -> str:
        return (f'Temperature({self._celsius:.2f}°C | '
                f'{self.fahrenheit:.2f}°F | {self.kelvin:.2f}K)')

t = Temperature(100)
print(t)                  # Temperature(100.00°C | 212.00°F | 373.15K)

t.celsius = 37            # викликає setter
print(t.fahrenheit)       # 98.6  - читається як атрибут
print(t.kelvin)           # 310.15

t.fahrenheit = 32         # setter fahrenheit → setter celsius
print(t.celsius)          # 0.0

# try:
#     t.celsius = -300    # ValueError!
# except ValueError as e:
#     print(e)

# ── Практичний приклад: валідований атрибут ──
class Person:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age  = age      # викликає setter!

    @property
    def age(self) -> int:
        return self._age

    @age.setter
    def age(self, value: int) -> None:
        if not isinstance(value, int) or value < 0 or value > 150:
            raise ValueError(f'Некоректний вік: {value}')
        self._age = value

    @property
    def is_adult(self) -> bool:    # обчислюваний, read-only
        return self._age >= 18

p = Person('Іван', 20)
print(p.is_adult)    # True
p.age = 25           # OK
# p.age = -5         # ValueError
# p.is_adult = True  # AttributeError: can't set attribute
```

Методи екземпляра, класові та статичні

Синтаксис / Концепція	Опис / Приклад
`def method(self)`	Метод екземпляра. self – посилання на об'єкт. Має доступ до self і cls через type(self).
`@classmethod def m(cls)`	Класовий метод. cls – посилання на сам клас. Не має доступу до конкретного self. Часто використовується як альтернативний конструктор.
`@staticmethod def m()`	Статичний метод. Не отримує ані self, ані cls. Це звичайна функція у просторі імен класу – утиліта, логічно пов'язана з класом.

```python
from datetime import date

class Student:
    _registry: list = []   # реєстр усіх студентів

    def __init__(self, name: str, birth_year: int, gpa: float):
        self.name       = name
        self.birth_year = birth_year
        self._gpa       = gpa
        Student._registry.append(self)

    # ── Метод екземпляра: працює з self ──
    def info(self) -> str:
        return f'{self.name}, {self.age} р., GPA: {self._gpa}'

    @property
    def age(self) -> int:
        return date.today().year - self.birth_year

    # ── Класовий метод: альтернативний конструктор ──
    @classmethod
    def from_string(cls, data: str) -> 'Student':
        '''Створює студента з рядка "Ім\'я,рік,GPA"'''
        name, year, gpa = data.split(',')
        return cls(name.strip(), int(year), float(gpa))

    @classmethod
    def get_registry(cls) -> list:
        return cls._registry.copy()

    @classmethod
    def get_count(cls) -> int:
        return len(cls._registry)

    # ── Статичний метод: утиліта ──
    @staticmethod
    def validate_gpa(gpa: float) -> bool:
        '''Перевіряє чи GPA у діапазоні 0..5'''
        return 0.0 <= gpa <= 5.0

    @staticmethod
    def gpa_to_grade(gpa: float) -> str:
        if gpa >= 4.5: return 'Відмінно'
        if gpa >= 3.5: return 'Добре'
        if gpa >= 2.5: return 'Задовільно'
        return 'Незадовільно'
```

```
    def __repr__(self) -> str:
        return f'Student({self.name!r})'

# Звичайне створення
s1 = Student('Іван', 2004, 4.2)

# Альтернативний конструктор через classmethod
s2 = Student.from_string('Марія, 2003, 4.8')

print(s1.info())               # Іван, 21 р., GPA: 4.2
print(s2.info())               # Марія, 22 р., GPA: 4.8
print(Student.get_count())     # 2

# Статичні методи – без self і cls
print(Student.validate_gpa(3.7))   # True
print(Student.validate_gpa(6.0))   # False
print(Student.gpa_to_grade(4.8))   # Відмінно
# Можна викликати і через екземпляр (але краще через клас)
print(s1.gpa_to_grade(2.3))        # Задовільно
```

Успадкування дозволяє новому класу (нащадку) отримати атрибути та методи батьківського класу і розширити або змінити їх. В Python кожен клас неявно успадковує від вбудованого `object`.

```
class Animal:
    '''Базовий клас для всіх тварин.'''

    def __init__(self, name: str, sound: str):
        self.name  = name
        self.sound = sound
        self._is_alive = True

    def speak(self) -> str:
        return f'{self.name} каже: {self.sound}!'

    def describe(self) -> str:
        return f'Тварина: {self.name}'

    def __repr__(self) -> str:
        return f'{self.__class__.__name__}({self.name!r})'
class Dog(Animal):    # Dog успадковує Animal
    def __init__(self, name: str, breed: str):
        super().__init__(name, 'Гав')   # виклик конструктора батька
        self.breed = breed

    def fetch(self) -> str:
        return f'{self.name} приносить м\'яч!'

    # Перевизначення (override) методу батька
    def describe(self) -> str:
        parent_desc = super().describe()  # виклик батьківського методу
        return f'{parent_desc}, порода: {self.breed}'

class Cat(Animal):
    def __init__(self, name: str, indoor: bool = True):
        super().__init__(name, 'Няв')
        self.indoor = indoor
```

```
    def purr(self) -> str:
        return f'{self.name} муркоче...'

    def speak(self) -> str:       # перевизначення
        base = super().speak()
        return f'{base} (і муркоче)'

class GuideDog(Dog):   # Дворівневе успадкування
    def __init__(self, name: str, breed: str, owner: str):
        super().__init__(name, breed)
        self.owner = owner


    def guide(self) -> str:
        return f'{self.name} веде {self.owner}'

    def describe(self) -> str:
        return f'{super().describe()}, поводир для: {self.owner}'

# ── Поліморфізм ──
animals = [Dog('Рекс', 'Вівчарка'), Cat('Мурка'), GuideDog('Арго', 'Лабрадор', 'Іван')]

for animal in animals:
    print(animal.speak())          # у кожного свій speak()
    print(animal.describe())
    print()

# Перевірка типів
dog = Dog('Бобік', 'Такса')
print(isinstance(dog, Dog))        # True
print(isinstance(dog, Animal))     # True  - успадкований тип
print(isinstance(dog, Cat))        # False
print(issubclass(Dog, Animal))     # True
print(issubclass(GuideDog, Dog))   # True
print(issubclass(GuideDog, Animal))# True
```

Функція `super()` та порядок вирішення методів (MRO)/
MRO (Method Resolution Order) – порядок, у якому Python шукає метод у ієрархії класів при множинному успадкуванні. Python використовує алгоритм C3 linearization.

Переглянути MRO: `ClassName.__mro__` або `ClassName.mro()`

```
# ── Множинне успадкування ──
class Flyable:
    def move(self) -> str:
        return 'Летить'
    def describe(self) -> str:
        return 'може літати'

class Swimmable:
    def move(self) -> str:
        return 'Пливе'
    def describe(self) -> str:
        return 'може плавати'

class Duck(Flyable, Swimmable):   # Flyable іде першим → його методи мають пріоритет
    def __init__(self, name: str):
        self.name = name
```

```
    def move(self) -> str:
        # super() у MRO шукає наступний клас у ланцюжку
        return f'{self.name}: {super().move()} і плаває'

duck = Duck('Кряк')
print(duck.move())          # Кряк: Летить і плаває
print(duck.describe())      # може літати  (Flyable – перший у MRO)

print(Duck.__mro__)
# (<class 'Duck'>, <class 'Flyable'>, <class 'Swimmable'>, <class 'object'>)

# ── Diamond problem та super() ──
class A:
    def hello(self):
        print('A.hello')

class B(A):
    def hello(self):
        super().hello()     # викличе C.hello (за MRO!)
        print('B.hello')

class C(A):
    def hello(self):
        super().hello()     # викличе A.hello
        print('C.hello')

class D(B, C):              # Diamond: D→B→C→A
    def hello(self):
        super().hello()
        print('D.hello')

print(D.__mro__)
# D → B → C → A → object

D().hello()
# A.hello  (спочатку A)
# C.hello  (потім C)
# B.hello  (потім B)
# D.hello  (нарешті D)
```

Правило `super()`!
Завжди використовуйте `super().__init__(...)` замість `ParentClass.__init__(self, ...)`.
Це гарантує правильну роботу MRO при множинному успадкуванні.
`super()` у методах завжди посилається на НАСТУПНИЙ клас у MRO, не обов'язково на безпосереднього батька.

Абстрактний клас – це клас, що визначає інтерфейс (набір методів), але не реалізує деякі з них. Він не може бути інстанційований напряму. Використовується для задання «контракту» для підкласів.

```python
from abc import ABC, abstractmethod
from math import pi

class Shape(ABC):     # успадковуємо ABC – Abstract Base Class
    '''Абстрактний базовий клас для геометричних фігур.'''

    def __init__(self, color: str = 'black'):
        self.color = color

    @abstractmethod
    def area(self) -> float:
        '''Обчислює площу фігури.'''
        ...

    @abstractmethod
    def perimeter(self) -> float:
        '''Обчислює периметр фігури.'''
        ...

    # Конкретний метод у абстрактному класі – дозволено
    def describe(self) -> str:
        return (f'{self.__class__.__name__} ({self.color}): '
                f'площа={self.area():.2f}, периметр={self.perimeter():.2f}')

    # Абстрактний classmethod
    @classmethod
    @abstractmethod
    def from_string(cls, s: str) -> 'Shape':
        ...

class Circle(Shape):
    def __init__(self, radius: float, color: str = 'red'):
        super().__init__(color)
        if radius <= 0:
            raise ValueError('Радіус повинен бути > 0')
        self.radius = radius

    def area(self) -> float:              # реалізуємо abstractmethod
        return pi * self.radius ** 2

    def perimeter(self) -> float:
        return 2 * pi * self.radius

    @classmethod
    def from_string(cls, s: str) -> 'Circle':
        r, color = s.split(',')
        return cls(float(r), color.strip())

    def __repr__(self) -> str:
        return f'Circle(r={self.radius}, color={self.color!r})'

class Rectangle(Shape):
    def __init__(self, width: float, height: float, color: str = 'blue'):
        super().__init__(color)
        self.width  = width
```

```
        self.height = height

    def area(self) -> float:
        return self.width * self.height

    def perimeter(self) -> float:
        return 2 * (self.width + self.height)

    @classmethod
    def from_string(cls, s: str) -> 'Rectangle':
        w, h, color = s.split(',')
        return cls(float(w), float(h), color.strip())

    def __repr__(self) -> str:
        return f'Rectangle({self.width}x{self.height})'

# shape = Shape()   # TypeError: Can't instantiate abstract class!

shapes: list[Shape] = [
    Circle(5, 'green'),
    Rectangle(4, 6, 'blue'),
    Circle.from_string('3, red'),
    Rectangle.from_string('10, 2, black'),
]

# Поліморфізм через спільний інтерфейс
for shape in shapes:
    print(shape.describe())

# Сортування за площею
sorted_shapes = sorted(shapes, key=lambda s: s.area())
print('\nВід меншої до більшої площі:')
for s in sorted_shapes:
    print(f'  {s}: {s.area():.2f}')
```

Декоратор `@dataclass` автоматично генерує `__init__`, `__repr__`, `__eq__` та інші методи на основі оголошених атрибутів. Ідеальний для класів-контейнерів даних.

```
from dataclasses import dataclass, field
from typing import ClassVar

@dataclass
class Point:
    x: float
    y: float

    def distance_to(self, other: 'Point') -> float:
        return ((self.x-other.x)**2 + (self.y-other.y)**2)**0.5

p1 = Point(1.0, 2.0)
p2 = Point(4.0, 6.0)
print(p1)                  # Point(x=1.0, y=2.0)  - автогенерований __repr__
print(p1 == Point(1.0, 2.0))  # True  - автогенерований __eq__
print(p1.distance_to(p2))  # 5.0

@dataclass(order=True)     # генерує <, <=, >, >= на основі порядку полів
class Student:
```

```
    # sort_index для сортування (не відображається у repr за замовч.)
    gpa:  float = field(compare=True)
    name: str   = field(compare=False)
    grades: list = field(default_factory=list, compare=False)

    # Атрибут класу (не екземпляра)
    _count: ClassVar[int] = 0

    def __post_init__(self):
        '''Викликається після автогенерованого __init__'''
        if not (0 <= self.gpa <= 5):
            raise ValueError(f'Невалідний GPA: {self.gpa}')
        Student._count += 1

students = [
    Student(4.5, 'Марія'),
    Student(3.8, 'Іван'),
    Student(4.9, 'Олена'),
]

# Сортування за GPA (order=True)
students.sort(reverse=True)
for s in students:
    print(s)
# Student(gpa=4.9, name='Олена', grades=[])
# Student(gpa=4.5, name='Марія', grades=[])
# Student(gpa=3.8, name='Іван',  grades=[])
```

Найкращі практики. `__slots__`, `__all__`, протоколи

`__slots__`: оптимізація пам'яті

```
# За замовчуванням атрибути зберігаються у __dict__ (словник)
# __slots__ замінює словник на фіксований набір - менше пам'яті

class PointSlots:
    __slots__ = ('x', 'y')   # дозволяємо лише ці атрибути

    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y

p = PointSlots(1.0, 2.0)
# p.z = 3.0   # AttributeError: 'PointSlots' has no attribute 'z'
# print(p.__dict__)  # AttributeError: немає __dict__!

# Для великої кількості об'єктів __slots__ економить ~40% пам'яті
```

Протокол Iterator: власний ітерований клас

```
class CountDown:
    '''Ітерований зворотній відлік.'''

    def __init__(self, start: int):
        self._start = start

    def __iter__(self):
        self._current = self._start
        return self

    def __next__(self):
```

```
        if self._current < 0:
            raise StopIteration
        value = self._current
        self._current -= 1
        return value

    def __len__(self):
        return self._start + 1

    def __contains__(self, item):
        return 0 <= item <= self._start

cd = CountDown(5)
print(list(cd))          # [5, 4, 3, 2, 1, 0]
print(len(cd))           # 6
print(3 in cd)           # True
for n in CountDown(3):   # 3 2 1 0
    print(n, end=' ')
```

Менеджер контексту через клас

```
class ManagedFile:
    '''Власний менеджер контексту для файлу.'''

    def __init__(self, filename: str, mode: str = 'r'):
        self.filename = filename
        self.mode     = mode
        self._file    = None

    def __enter__(self):
        print(f'Відкриваємо: {self.filename}')
        self._file = open(self.filename, self.mode, encoding='utf-8')
        return self._file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self._file:
            self._file.close()
        print(f'Закрито: {self.filename}')
        # Повертаємо False: не пригнічуємо виняток
        return False

with ManagedFile('data.txt', 'w') as f:
    f.write('Тест\n')
# Відкриваємо: data.txt
# Закрито: data.txt
```

Комплексний приклад: система управління бібліотекою.

Розглянемо повноцінний приклад застосування всіх концепцій ООП у реальному сценарії.

```
from abc import ABC, abstractmethod
from dataclasses import dataclass, field
from datetime import date, timedelta
from typing import Optional

# ── Абстрактний базовий клас ──
class LibraryItem(ABC):
    _total_items: int = 0

    def __init__(self, title: str, year: int):
```

```
        self.title      = title
        self.year       = year
        self._available = True
        LibraryItem._total_items += 1

    @abstractmethod
    def get_info(self) -> str: ...

    @abstractmethod
    def loan_period_days(self) -> int: ...

    @property
    def available(self) -> bool:
        return self._available

    def checkout(self) -> bool:
        if not self._available:
            return False
        self._available = False
        return True

    def return_item(self) -> None:
        self._available = True

    @classmethod
    def total_items(cls) -> int:
        return cls._total_items

    def __repr__(self) -> str:
        status = 'доступна' if self._available else 'видана'
        return f'{self.__class__.__name__}({self.title!r}) [{status}]'

class Book(LibraryItem):
    def __init__(self, title: str, author: str, year: int, pages: int):
        super().__init__(title, year)
        self.author = author
        self.pages  = pages

    def get_info(self) -> str:
        return f'Книга: "{self.title}" – {self.author} ({self.year}, {self.pages}
стор.)'

    def loan_period_days(self) -> int:
        return 14   # 2 тижні

class DVD(LibraryItem):
    def __init__(self, title: str, director: str, year: int, duration_min: int):
        super().__init__(title, year)
        self.director     = director
        self.duration_min = duration_min

    def get_info(self) -> str:
        return f'DVD: "{self.title}" – реж. {self.director} ({self.duration_min} хв.)'

    def loan_period_days(self) -> int:
        return 3    # 3 дні

@dataclass
class Loan:
```

```
    item:      LibraryItem
    borrower:  str
    loan_date: date = field(default_factory=date.today)

    @property
    def due_date(self) -> date:
        return self.loan_date + timedelta(days=self.item.loan_period_days())

    @property
    def is_overdue(self) -> bool:
        return date.today() > self.due_date

    def __str__(self) -> str:
        status = ' [ПРОСТРОЧЕНО!]' if self.is_overdue else ''
        return (f'{self.borrower}: "{self.item.title}"'
                f' до {self.due_date}{status}')

class Library:
    def __init__(self, name: str):
        self.name    = name
        self._items: list[LibraryItem] = []
        self._loans: list[Loan]        = []

    def add_item(self, item: LibraryItem) -> None:
        self._items.append(item)

    def checkout(self, title: str, borrower: str) -> Optional[Loan]:
        for item in self._items:
            if item.title == title and item.available:
                item.checkout()
                loan = Loan(item, borrower)
                self._loans.append(loan)
                return loan
        return None

    def return_item(self, title: str) -> bool:
        for loan in self._loans:
            if loan.item.title == title and not loan.item.available:
                loan.item.return_item()
                self._loans.remove(loan)
                return True
        return False

    def available_items(self) -> list[LibraryItem]:
        return [i for i in self._items if i.available]

    def report(self) -> None:
        print(f'=== Бібліотека "{self.name}" ===')
        print(f'Всього одиниць: {len(self._items)}')
        print(f'Доступних:      {len(self.available_items())}')
        print(f'Виданих:        {len(self._loans)}')
        if self._loans:
            print('Активні позики:')
            for loan in self._loans:
                print(f'  {loan}')

# ── Використання ──
lib = Library('Міська бібліотека')
```

```
lib.add_item(Book('Кобзар', 'Т. Шевченко', 1840, 312))
lib.add_item(Book('Тіні забутих предків', 'М. Коцюбинський', 1911, 128))
lib.add_item(DVD('Поводир', 'О. Саніна', 2013, 112))

loan1 = lib.checkout('Кобзар', 'Іван Петренко')
loan2 = lib.checkout('Поводир', 'Марія Коваль')

lib.report()

lib.return_item('Кобзар')
print('\nПісля повернення:')
lib.report()
```

Завдання на лабораторну роботу

Кожне завдання оформляйте у окремому файлі (task1.py ... task3.py). Для класів обов'язково пишіть docstrings. Використовуйте анотації типів.

8.1. Клас з атрибутами, методами та `@property`. Розробіть клас `Rectangle` (прямокутник):

- конструктор приймає `width` та `height`. Обидва – protected через `@property` з валідацією (> 0);
- Properties (read-only): `area`, `perimeter`, `diagonal`, `is_square`;
- методи: `scale(factor)` – повертає новий `Rectangle` із масштабованими розмірами; `describe()` – форматований рядок опису;
- спеціальні методи: `__repr__`, `__str__`, `__eq__` (рівність за розмірами), `__lt__` (порівняння за площею), `__add__` (повертає прямокутник із сумарною площею, що має ту ж ширину);
- класовий метод `from_string('4x5')` та статичний метод `is_valid(w, h)`.

Напишіть мінімум 10 тестових виразів, що демонструють усі можливості класу.

8.2. Інкапсуляція. Public, protected, private. Розробіть клас `Student` із такою структурою атрибутів:

- Public: `name`, `year` (рік навчання 1–6);
- Protected: `_gpa` (середній бал), `_courses` (список курсів);
- Private: `__student_id` (автогенерований номер: `'STU-001'`, `'STU-002'` тощо);
- `@property` з getter та setter для `gpa` (валідація 0–5) та `year` (валідація 1–6);
- методи: `add_course(name)`, `remove_course(name)`, `get_transcript()` → форматована відомість;
- класові методи: `get_count()`, `find_by_id(id)` → повертає студента з реєстру;
- статичний метод: `validate_name(name)` → перевіряє що ім'я складається лише з літер.

Продемонструйте спробу прямого доступу до `__student_id` та порівняйте з правильним способом.

8.3. Успадкування та поліморфізм.

Розробіть ієрархію класів для системи транспортних засобів:

- Базовий клас `Vehicle`: `make`, `model`, `year`, `_mileage`; методи `drive(km)`, `refuel(liters)`, `status()`; `@property mileage` (read-only); `__repr__`;
- Клас `Car(Vehicle)`: додає `fuel_type`, `fuel_level`; перевизначає `drive()` (зменшує паливо); додає `property range` (запас ходу);

- Клас `ElectricCar(Car)`: замість палива – `battery_level` (%); перевизначає `drive()` та `refuel()` (перейменувати на `charge()`);
- Клас `Truck(Vehicle)`: вантажопідйомність, поточна вага вантажу; методи `load(kg)` та `unload(kg)` з перевіркою;
- Клас `FleetManager`: зберігає список `Vehicle`; методи `add_vehicle`, `remove_vehicle`, `total_mileage`, `find_by_make(make)`, `report()`.

Створіть флот з 5–6 різних автомобілів. Продемонструйте поліморфізм: для кожного транспорту викличте `drive(100)` – у кожного своя логіка.

8.4. Вбудовані-методи та протоколи. Розробіть клас Matrix (математична матриця):

- `__init__(rows, cols, data=None)`: якщо data не задано – заповнити нулями;
- `__repr__, __str__` (форматований вивід у вигляді таблиці з вирівнюванням)
- `__eq__, __add__, __sub__, __mul__` (матричне множення `@` або scalar `*`)
- `__getitem__(self, key)` де `key` – кортеж `(i, j)` і `__setitem__`
- `__iter__` – ітерація по рядках матриці
- `__len__` – кількість рядків
- `Property: rows, cols, is_square, trace` (слід – сума діагональних елементів)
- Методи: `transpose(), identity(n)` (`classmethod` – одинична матриця)

Продемонструйте всі операції, включаючи: `m[0,1] = 5, for row in matrix, len(matrix)`.

8.5. Комплексна ієрархія класів зі всіма концепціями. Розробіть систему управління університетом. Ієрархія класів:

- Абстрактний клас `Person(ABC)`: `name, email, __id` (private, автогенерований); abstract методи `role(), contact_info(); dunder __eq__, __hash__, __repr__`;
- Клас `Student(Person)`: успадковує `Person`, додає `year, major, _grades (dict {subject: [оцінки]})`; методи `add_grade(subject, grade), gpa (property), academic_status` (property: 'відмінник'/'хорошист' тощо), `transcript();`
- Клас `Teacher(Person): department, _salary` (property з setter з валідацією), `courses (list)`; методи `assign_course, remove_course, annual_salary (property);`
- Клас `GradStudent(Student, Teacher)`: успадковує обидва; додає `thesis_topic, supervisor`; перевизначає `role()` та `contact_info()`; демонструє правильне використання `super()` у MRO;
- Клас `Department`: `name, head (Teacher), members (list[Person])`; методи `add_member, remove_member, average_student_gpa, teacher_count, student_count, report();`
- Клас `University`: `name, _departments (dict)`; методи `add_department, enroll_student(dept_name, student), hire_teacher(dept_name, teacher), overall_stats(), find_person(name).`

Додаткові вимоги: використайте `@dataclass` для допоміжного класу `Grade(subject, value, date)`; реалізуйте `__slots__` у Person для оптимізації; додайте ітерацію по `Department` (for person in dept); збережіть список усіх осіб у реєстрі класу `Person`. Створіть університет із 2 кафедрами, 8 студентами, 4 викладачами та 1 аспірантом. Виведіть повний звіт.

Контрольні запитання

1. Що таке клас і об'єкт у Python? Поясніть різницю між атрибутом класу та атрибутом екземпляра на конкретному прикладі.
2. Поясніть механізм інкапсуляції в Python. Яка різниця між `_name` та `__name`? Що таке name mangling і навіщо він потрібен?
3. Що таке `@property`? Як реалізувати атрибут лише для читання (read-only) через `@property`? Яку проблему вирішує `@property` порівняно з прямим доступом до атрибута?
4. У чому різниця між методом екземпляра, `@classmethod` та `@staticmethod`? Коли доцільно використовувати кожен з них?
5. Що таке `__init__` і `__new__`? Що таке спеціальні методи? Назвіть п'ять спеціальних методів, поясніть які вбудовані операції вони підтримують.
6. Поясніть механізм успадкування. Для чого використовується `super()`? Що відбудеться, якщо замінити `super().__init__()` на `ParentClass.__init__(self)`?
7. Що таке MRO (Method Resolution Order)? За яким алгоритмом Python визначає порядок пошуку методів при множинному успадкуванні? Яку послідовність класів поверне `Dog.__mro__` для class `Dog(Flyable, Swimmable)`?
8. Що таке абстрактний клас? Навіщо використовується модуль `abc`? Що станеться, якщо спробувати створити екземпляр класу, що успадковує ABC, але не реалізує всі `@abstractmethod`?
9. Що таке `@dataclass`? Які спеціальні методи він генерує автоматично? Навіщо потрібен `__post_init__`?
10. Що таке `__slots__`? Яку перевагу дає його використання? В яких ситуаціях воно особливо корисне?

Лабораторна робота №8.

Unit-тестування програмного коду на Python

Мета роботи: Зрозуміти призначення та принципи автоматичного тестування програмного забезпечення, навчитися писати unit-тести за допомогою бібліотеки pytest: перевіряти правильність функцій, методів класів та обробку виняткових ситуацій. Опанувати механізм фікстур (fixtures) для підготовки та прибирання тестового середовища, навчитися використовувати параметризацію тестів для перевірки різних вхідних даних без дублювання коду. Ознайомитися з mock-об'єктами та методами їх застосування для ізоляції тесту від зовнішніх залежностей; навчитися вимірювати покриття коду (code coverage) та розуміти, чому 100% покриття є важливою, але не достатньою метрикою якості.

Теоретичні відомості:

Тест – це код, який перевіряє інший код. Автоматичні тести виконуються миттєво, повторювано і без участі людини. Без тестів кожна зміна в програмі потенційно ламає щось непомітно.

Піраміда тестування описує три рівні тестів за кількістю та швидкістю:

- Unit-тести (нижній, найбільший рівень) – перевіряють окрему функцію або метод в ізоляції. Найшвидші та найдешевші у написанні й виконанні.
- Інтеграційні тести (середній рівень) – перевіряють взаємодію кількох компонентів: функція + база даних, клас + файлова система тощо.
- End-to-end тести (верхній, найменший рівень) – перевіряють всю систему від початку до кінця, як реальний користувач. Найповільніші та найдорожчі.

Правило хорошого unit-тесту (FIRST):

F – Fast (Швидкий): тест виконується за мілісекунди.
I – Independent (Незалежний): тести не залежать один від одного.
R – Repeatable (Відтворюваний): однаковий результат за будь-яких умов.
S – Self-validating (Самоперевіряючий): або зелений, або червоний.
T – Timely (Своєчасний): пишеться разом із кодом, а не після.

Встановлення pytest. Базовий синтаксис

pytest – найпопулярніша бібліотека тестування для Python. Вона розширює вбудований unittest та пропонує значно лаконічніший синтаксис.

```
# Встановлення
pip install pytest pytest-cov

# Запуск тестів
pytest                          # всі тести у поточній директорії
pytest test_math.py             # конкретний файл
pytest test_math.py::test_add # конкретна функція
pytest -v                       # verbose: показує назву кожного тесту
pytest -v -s                    # також показує print() всередині тестів
pytest -k 'add or subtract'     # тести, чиї імена містять 'add' або 'subtract'
pytest --tb=short               # короткий traceback при помилках
pytest -x                       # зупинитись після першого провалу
```

Структура тестового файлу – файл повинен називатись `test_*.py` або `*_test.py`, функції – починатись з `test_`:

```python
# — Код, який тестуємо (зазвичай у окремому файлі) —
# math_utils.py
def add(a, b):
    return a + b

def divide(a, b):
    if b == 0:
        raise ZeroDivisionError('Ділення на нуль!')
    return a / b

def is_even(n):
    return n % 2 == 0
```

```python
# — Тестовий файл —
from math_utils import add, divide, is_even

# Проста перевірка через assert
def test_add_positive_numbers():
    assert add(2, 3) == 5

def test_add_negative_numbers():
    assert add(-1, -2) == -3

def test_add_zero():
    assert add(5, 0) == 5

def test_is_even_true():
    assert is_even(4) is True

def test_is_even_false():
    assert is_even(7) is False

# Перевірка, що функція кидає виняток
def test_divide_by_zero():
    import pytest
    with pytest.raises(ZeroDivisionError):
        divide(10, 0)

def test_divide_by_zero_message():
    import pytest
    with pytest.raises(ZeroDivisionError, match='Ділення на нуль'):
        divide(5, 0)

def test_divide_normal():
    result = divide(10, 4)
    assert result == pytest.approx(2.5)   # для дробових чисел!
```

Важливо: pytest.approx()

Ніколи не порівнюйте дробові числа через ==!

`assert 0.1 + 0.2 == 0.3` → *ПРОВАЛ (через похибку float)*

`assert 0.1 + 0.2 == pytest.approx(0.3)` → *УСПІХ*

`pytest.approx` *має допустиму похибку 1e-6 за замовчуванням.*

pytest перехоплює стандартний assert і надає детальне повідомлення при провалі – без додаткових зусиль з боку розробника. Це головна перевага перед unittest.

```
# При провалі pytest показує КОНКРЕТНО, що не збігається

def test_list_content():
    result = [1, 2, 3, 4]
    expected = [1, 2, 3, 5]
    assert result == expected
# FAIL: assert [1, 2, 3, 4] == [1, 2, 3, 5]
#   At index 3 diff: 4 != 5

def test_string_content():
    s = 'hello world'
    assert 'python' in s
# FAIL: assert 'python' in 'hello world'

# Власне повідомлення при провалі
def test_with_message():
    age = 15
    assert age >= 18, f'Очікувався вік >= 18, отримано {age}'

# Перевірка типу
def test_return_type():
    result = add(2, 3)
    assert isinstance(result, int), f'Очікувався int, отримано {type(result)}'

# Перевірка довжини
def test_list_length():
    items = get_items()    # якась функція
    assert len(items) == 3, f'Очікувалось 3 елементи, отримано {len(items)}'

# Перевірка кількох умов в одному тесті
def test_circle_properties():
    c = Circle(5)
    assert c.area()      == pytest.approx(78.539, rel=1e-3)
    assert c.perimeter() == pytest.approx(31.416, rel=1e-3)
    assert c.radius == 5
```

`@pytest.mark.parametrize` дозволяє запустити один тест з різними наборами вхідних даних. Це усуває дублювання коду та дає зрозумілий звіт для кожного набору окремо.

```
import pytest
from math_utils import add, divide, is_even

# ── Базова параметризація ──
@pytest.mark.parametrize('a, b, expected', [
    (2,   3,   5),
    (-1, -2,  -3),
    (0,   0,   0),
    (100, -50, 50),
])
def test_add(a, b, expected):
    assert add(a, b) == expected
# pytest генерує 4 окремих тести: test_add[2-3-5], test_add[-1--2--3] ...

# ── Параметризація з очікуваним True/False ──
@pytest.mark.parametrize('number, expected', [
```

```
    (2,  True),
    (3,  False),
    (0,  True),
    (-4, True),
    (-7, False),
])
def test_is_even(number, expected):
    assert is_even(number) == expected

# ── Параметризація з іменованими варіантами (ids) ──
@pytest.mark.parametrize('text, expected', [
    ('hello',       'HELLO'),
    ('world',       'WORLD'),
    ('',            ''),
    ('ALREADY UP', 'ALREADY UP'),
], ids=['lower', 'single', 'empty', 'already_upper'])
def test_uppercase(text, expected):
    assert text.upper() == expected

# ── Параметризація з очікуваними винятками ──
@pytest.mark.parametrize('a, b, exc_type', [
    (10,  0,    ZeroDivisionError),
    ('x', 1,    TypeError),
])
def test_divide_errors(a, b, exc_type):
    with pytest.raises(exc_type):
        divide(a, b)

# ── Двовимірна параметризація ──
@pytest.mark.parametrize('n',      [2, 3, 5])
@pytest.mark.parametrize('factor', [1, 2])
def test_multiply_table(n, factor):
    # 6 комбінацій: (2,1),(3,1),(5,1),(2,2),(3,2),(5,2)
    result = n * factor
    assert isinstance(result, int)
    assert result > 0
```

Фікстура – функція, що готує тестове середовище і може прибирати за собою після тесту. Вона вирішує проблему дублювання підготовчого коду між тестами.

Область видимості (scope) фікстури визначає, як часто вона виконується: function (за замовчуванням), class, module, session.

```
import pytest
from bank import BankAccount

# ── Базова фікстура (scope='function' за замовчуванням) ──
@pytest.fixture
def account():
    '''Створює новий рахунок перед кожним тестом.'''
    return BankAccount('Іван', 1000.0)

# Тест отримує фікстуру як аргумент (ін'єкція залежностей)
def test_initial_balance(account):
    assert account._balance == 1000.0

def test_deposit(account):
    account.deposit(500)
    assert account._balance == 1500.0
```

```python
def test_withdraw(account):
    success = account.withdraw(200, '0000')
    assert success is True
    assert account._balance == 800.0

def test_overdraft(account):
    success = account.withdraw(2000, '0000')
    assert success is False
    assert account._balance == 1000.0  # не змінився

# — Фікстура з прибиранням (yield) —
@pytest.fixture
def temp_file(tmp_path):
    '''Створює тимчасовий файл; tmp_path – вбудована фікстура pytest.'''
    filepath = tmp_path / 'test_data.txt'
    filepath.write_text('рядок1\nрядок2\nрядок3', encoding='utf-8')
    yield filepath           # передаємо файл у тест
    # Код після yield – це teardown (прибирання)
    # tmp_path прибирається автоматично, але можна додати свою логіку
    print(f'Тест з файлом {filepath.name} завершено')

def test_file_line_count(temp_file):
    lines = temp_file.read_text(encoding='utf-8').splitlines()
    assert len(lines) == 3

def test_file_content(temp_file):
    content = temp_file.read_text(encoding='utf-8')
    assert 'рядок1' in content

# — Фікстура з широкою областю видимості (module) —
@pytest.fixture(scope='module')
def db_connection():
    '''Підключення до бази – один раз на модуль (не на кожен тест).'''
    import sqlite3
    conn = sqlite3.connect(':memory:')   # тимчасова БД у пам'яті
    conn.execute('CREATE TABLE users (id INTEGER, name TEXT)')
    conn.execute("INSERT INTO users VALUES (1, 'Іван')")
    conn.commit()
    yield conn
    conn.close()   # teardown: закриваємо після всього модуля

def test_db_has_user(db_connection):
    cursor = db_connection.execute('SELECT name FROM users WHERE id=1')
    row = cursor.fetchone()
    assert row[0] == 'Іван'

# — conftest.py: спільні фікстури для всього проєкту —
# Фікстури у файлі conftest.py доступні у всіх тестових файлах
# автоматично, без явного import!
```

Маркери. Пропуск тестів та очікувані провали

Синтаксис / Концепція	Опис / Результат
`@pytest.mark.skip`	Завжди пропускає тест. Параметр reason.
`@pytest.mark.skipif(cond)`	Пропускає тест якщо умова cond == True.
`@pytest.mark.xfail`	Очікуваний провал. Якщо провалиться – XFAIL (норма). Якщо пройде – XPASS (несподівано).
`@pytest.mark.slow`	Власний маркер (потрібна реєстрація у pytest.ini).

`@pytest.mark.integration`	Власний маркер для інтеграційних тестів.
`pytest -m 'not slow'`	Запустити всі тести, крім позначених 'slow'.
`pytest -m 'unit'`	Запустити лише тести з маркером 'unit'.

```python
import pytest
import sys

# ── Пропуск тесту ──
@pytest.mark.skip(reason='Функція ще не реалізована')
def test_new_feature():
    assert new_feature() == 42

# ── Умовний пропуск ──
@pytest.mark.skipif(sys.platform == 'win32',
                    reason='Тест не підтримується на Windows')
def test_unix_permissions():
    import os
    assert os.access('/etc/passwd', os.R_OK)

# ── Очікуваний провал ──
@pytest.mark.xfail(reason='Відомий баг #123, чекаємо на виправлення')
def test_known_bug():
    assert broken_function() == 'correct'

@pytest.mark.xfail(strict=True)  # strict=True: XPASS → помилка
def test_must_fail():
    assert 1 == 2

# ── Власні маркери (конфігурація у pytest.ini) ──
# pytest.ini:
# [pytest]
# markers =
#     slow: marks tests as slow (deselect with '-m "not slow"')
#     integration: marks tests as integration tests
#     unit: marks tests as unit tests

@pytest.mark.slow
@pytest.mark.integration
def test_heavy_computation():
    import time
    time.sleep(0.1)    # повільна операція
    result = sum(range(1_000_000))
    assert result == 499999500000
```

Mock (підробка) – це об'єкт, що замінює реальну залежність у тесті. Він дозволяє тестувати код в ізоляції: без мережі, без бази даних, без зовнішніх сервісів, без файлової системи. Стандартна бібліотека: `unittest.mock`. Ключові класи: `Mock, MagicMock, patch`.

```python
from unittest.mock import Mock, MagicMock, patch, call
import pytest

# ── Базовий Mock ──
mock = Mock()
mock.some_method(1, 2, 3)    # будь-який виклик приймається
mock.some_attr = 42

print(mock.some_method.called)          # True
```

```python
print(mock.some_method.call_count)       # 1
print(mock.some_method.call_args)        # call(1, 2, 3)

# — Mock із заданим return_value —
def get_user_age(user_service, user_id):
    '''Функція, яку ми тестуємо.'''
    user = user_service.get_user(user_id)
    return user['age']

def test_get_user_age():
    # Замінюємо user_service на Mock
    mock_service = Mock()
    mock_service.get_user.return_value = {'name': 'Іван', 'age': 21}

    result = get_user_age(mock_service, 1)

    assert result == 21
    mock_service.get_user.assert_called_once_with(1)  # перевірка виклику

# — Mock що кидає виняток —
def test_service_unavailable():
    mock_service = Mock()
    mock_service.get_user.side_effect = ConnectionError('Сервер недоступний')

    with pytest.raises(ConnectionError):
        get_user_age(mock_service, 1)

# — patch: заміна об'єкта у конкретному модулі —
# Тестуємо функцію, що використовує datetime.now()

# weather.py
# import datetime
# def get_greeting():
#     hour = datetime.datetime.now().hour
#     if hour < 12: return 'Доброго ранку!'
#     if hour < 18: return 'Доброго дня!'
#     return 'Доброго вечора!'

from unittest.mock import patch
import datetime

@patch('weather.datetime')
def test_morning_greeting(mock_dt):
    mock_dt.datetime.now.return_value = datetime.datetime(2024, 1, 1, 9, 0)
    from weather import get_greeting
    assert get_greeting() == 'Доброго ранку!'

@patch('weather.datetime')
def test_evening_greeting(mock_dt):
    mock_dt.datetime.now.return_value = datetime.datetime(2024, 1, 1, 20, 0)
    from weather import get_greeting
    assert get_greeting() == 'Доброго вечора!'

# — patch як контекстний менеджер —
def test_file_reading():
    mock_content = 'рядок1\nрядок2\nрядок3'
    with patch('builtins.open', unittest.mock.mock_open(read_data=mock_content)):
        from file_processor import count_lines
        assert count_lines('any_file.txt') == 3
```

При тестуванні класів зручно використовувати клас тестів (TestClass) для групування пов'язаних тестів. Кожен метод класу починається з test_.

```python
import pytest
from shapes import Circle, Rectangle  # класи з лабораторної роботи №8

class TestCircle:
    '''Тести для класу Circle.'''

    @pytest.fixture(autouse=True)
    def setup(self):
        '''autouse=True: фікстура застосовується до всіх методів класу.'''
        self.circle = Circle(5)

    def test_initial_radius(self):
        assert self.circle.radius == 5

    def test_area(self):
        assert self.circle.area() == pytest.approx(78.539, rel=1e-3)

    def test_perimeter(self):
        assert self.circle.perimeter() == pytest.approx(31.416, rel=1e-3)

    def test_negative_radius_raises(self):
        with pytest.raises(ValueError, match='> 0'):
            Circle(-1)

    def test_zero_radius_raises(self):
        with pytest.raises(ValueError):
            Circle(0)

    @pytest.mark.parametrize('radius, expected_area', [
        (1,   pytest.approx(3.14159, rel=1e-4)),
        (2,   pytest.approx(12.566,  rel=1e-4)),
        (10,  pytest.approx(314.159, rel=1e-4)),
])
    def test_area_parametrized(self, radius, expected_area):
        c = Circle(radius)
        assert c.area() == expected_area


class TestRectangle:
    def test_area(self):
        r = Rectangle(4, 6)
        assert r.area() == 24

    def test_perimeter(self):
        r = Rectangle(3, 5)
        assert r.perimeter() == 16

    def test_is_square_true(self):
        r = Rectangle(5, 5)
        assert r.is_square is True

    def test_is_square_false(self):
        r = Rectangle(4, 6)
        assert r.is_square is False
```

```python
    def test_scale(self):
        r = Rectangle(4, 6)
        scaled = r.scale(2)
        assert scaled.width  == 8
        assert scaled.height == 12
        # Оригінал не змінився
        assert r.width  == 4
        assert r.height == 6
```

Покриття коду (coverage) показує, який відсоток рядків коду виконується під час тестів. Інструмент pytest-cov вимірює покриття та генерує звіт.

```
# Встановлення
pip install pytest-cov

# Запуск із вимірюванням покриття
pytest --cov=my_module test_my_module.py

# Звіт у термінал з переліком непокритих рядків
pytest --cov=my_module --cov-report=term-missing

# HTML-звіт (відкрити htmlcov/index.html у браузері)
pytest --cov=my_module --cov-report=html

# Мінімальний поріг покриття (CI/CD не пропустить < 80%)
pytest --cov=my_module --cov-fail-under=80

# Покриття всього проєкту
pytest --cov=. --cov-report=term-missing

# ── Типовий вивід ──
# Name               Stmts   Miss  Cover   Missing
# ------------------------------------------------
# math_utils.py         12      2    83%   15, 23
# bank.py               25      3    88%   42-44
# ------------------------------------------------
# TOTAL                 37      5    86%

# ── Конфігурація у .coveragerc або pyproject.toml ──
# [coverage:run]
# source = .
# omit =
#     */tests/*
#     */migrations/*
#     conftest.py
#
# [coverage:report]
# fail_under = 80
# show_missing = True
```

Coverage != якість тестів!
100% покриття НЕ означає, що тести хороші.
Код може виконуватись, але не перевірятись на правильність.
Мета покриття – знайти НЕПОКРИТІ ділянки, а не отримати 100%.
Краща метрика: `mutation testing` *(перевірка чи тести помічають навмисні помилки).*

Структура тестового проєкту. pytest.ini та conftest.py
Правильна організація тестів спрощує підтримку та масштабування проєкту.

```
# Рекомендована структура проєкту
my_project/
├── src/                    # або просто корінь проєкту
│   ├── math_utils.py
│   ├── bank.py
│   └── shapes.py
├── tests/
│   ├── conftest.py         # спільні фікстури для всього проєкту
│   ├── unit/
│   │   ├── test_math_utils.py
│   │   ├── test_bank.py
│   │   └── test_shapes.py
│   └── integration/
│       └── test_database.py
├── pytest.ini              # конфігурація pytest
└── requirements.txt

# ── pytest.ini ──
[pytest]
testpaths = tests
addopts = -v --tb=short
markers =
    unit: unit tests
    integration: integration tests
    slow: slow tests (deselect with '-m "not slow"')


# ── tests/conftest.py ── (спільні фікстури)
import pytest
from bank import BankAccount
from shapes import Circle

@pytest.fixture(scope='session')
def config():
    '''Конфігурація для всієї сесії тестів.'''
    return {'max_balance': 1_000_000, 'min_age': 18}

@pytest.fixture
def sample_account():
    return BankAccount('Тестовий користувач', 500.0)

@pytest.fixture
def sample_shapes():
    return [Circle(1), Circle(5), Circle(10)]


# ── Запуск тільки unit-тестів ──
# pytest -m unit

# ── Запуск без slow тестів ──
# pytest -m 'not slow'

# ── Паралельний запуск (потрібен pytest-xdist) ──
# pip install pytest-xdist
# pytest -n 4   # 4 паралельних процеси
```

TDD (Test-Driven Development) – підхід, де тест пишеться ДО реалізації коду. Цикл TDD: Red → Green → Refactor.

- Red – написати тест, що провалюється (код ще не існує)
- Green – написати мінімальний код, що робить тест зеленим
- Refactor – покращити код без зміни поведінки (тести гарантують це)

```
# Демонстрація TDD для класу Stack

# ── Крок 1: RED – пишемо тести перед кодом ──

# test_stack.py
import pytest
# from stack import Stack  # поки не існує!

class TestStack:
    def test_new_stack_is_empty(self):
        s = Stack()
        assert s.is_empty() is True
        assert len(s) == 0

    def test_push_increases_size(self):
        s = Stack()
        s.push(1)
        assert len(s) == 1
        assert s.is_empty() is False

    def test_pop_returns_last(self):
        s = Stack()
        s.push(1)
        s.push(2)
        s.push(3)
        assert s.pop() == 3
        assert s.pop() == 2

    def test_peek_does_not_remove(self):
        s = Stack()
        s.push(42)
        assert s.peek() == 42
        assert len(s) == 1   # peek не видаляє!

    def test_pop_empty_raises(self):
        s = Stack()
        with pytest.raises(IndexError, match='порожній'):
            s.pop()


# ── Крок 2: GREEN – мінімальна реалізація ──

# stack.py
class Stack:
    def __init__(self):
        self._data = []

    def push(self, item) -> None:
        self._data.append(item)

    def pop(self):
```

```
        if self.is_empty():
            raise IndexError('Стек порожній')
        return self._data.pop()

    def peek(self):
        if self.is_empty():
            raise IndexError('Стек порожній')
        return self._data[-1]

    def is_empty(self) -> bool:
        return len(self._data) == 0

    def __len__(self) -> int:
        return len(self._data)

# ── Крок 3: REFACTOR – покращуємо, тести гарантують коректність ──
# Додаємо __repr__, type hints, docstrings – тести залишаються зеленими
```

Завдання на лабораторну роботу

Для кожного завдання створіть окрему пару файлів: код (наприклад, calculator.py) та тести (test_calculator.py). Усі завдання виконуйте у спільній директорії lab9/ зі спільним conftest.py та pytest.ini.
Перед виконанням: pip install pytest pytest-cov

8.1. Напишіть файл math_utils.py із такими функціями:
- `factorial(n)` – факторіал (рекурсивно); кидає `ValueError` при n < 0;
- `is_prime(n)` – перевірка на простоту; кидає `ValueError` при n < 2;
- `gcd(a, b)` – НСД (найбільший спільний дільник) алгоритмом Евкліда;
- `clamp(value, lo, hi)` – затискає значення у діапазоні `[lo, hi]`.

Напишіть файл `test_math_utils.py` із мінімум 20 тестами, що покривають:
- нормальні випадки для кожної функції (мінімум 3 параметризованих набори);
- граничні значення (0, 1, від'ємні числа, дуже великі числа);
- перевірку винятків через `pytest.raises`;
- перевірку типу результату.

Виміряйте покриття коду: `pytest --cov=math_utils --cov-report=term-missing`

8.2. Напишіть файл `string_utils.py` із функціями:
- `is_palindrome(s)` – чи є рядок паліндромом (без урахування регістру та пробілів);
- `count_words(text)` – кількість слів у тексті;
- `truncate(text, max_len, suffix='...')` – обрізає рядок до `max_len` символів, додаючи `suffix`;
- `to_snake_case(s)` – перетворює 'Hello World' або 'helloWorld' у 'hello_world'

Напишіть тести з обов'язковим використанням `@pytest.mark.parametrize` для кожної функції. Кожна параметризація повинна мати мінімум 5 наборів даних, включаючи порожні рядки та граничні випадки.

8.3 Візьміть клас `BankAccount` із теоретичних відомостей (або реалізуйте власний). Напишіть тестовий файл `test_bank.py`, що використовує:

- фікстуру `account()` – порожній рахунок;
- фікстуру `funded_account()` – рахунок з балансом 1000 грн і кількома транзакціями;
- фікстуру `tmp_path` для тесту, що зберігає виписку рахунку у файл;

Тести повинні перевіряти: початковий стан, поповнення, зняття, неправильний PIN, спробу зняти більше ніж є, журнал операцій. Згрупуйте тести у клас TestBankAccount.

8.4. Напишіть файл validators.py із функціями валідації:

- `validate_email(email) → bool` – перевіряє формат email через регулярний вираз;
- `validate_phone(phone) → bool` – перевіряє формат українського телефону
- `validate_password(pwd) → dict` – повертає `{'valid': bool, 'errors': [список помилок]}`, де пароль вважається валідним якщо: довжина >= 8, є велика літера, є цифра, є спецсимвол
- `validate_date(s) → bool` – перевіряє формат 'DD.MM.YYYY' та реальність дати

Напишіть тести з повною параметризацією: мінімум 8 валідних і 8 невалідних значень для кожної функції. Позначте «важкі» тести з великими наборами даних маркером `@pytest.mark.slow`.

Контрольні запитання

1. Що таке unit-тест? Чим він відрізняється від інтеграційного та end-to-end тесту? Поясніть піраміду тестування.
2. Які вимоги FIRST висуваються до хорошого unit-тесту? Поясніть кожну вимогу.
3. Яка структура файлів і функцій потрібна для того, щоб pytest автоматично знаходив тести? Що станеться, якщо назвати тестову функцію check_addition замість test_addition?
4. Чому не можна порівнювати дробові числа через == у тестах? Як правильно порівнювати результати обчислень з плаваючою крапкою?
5. Що таке фікстура у pytest? Які значення може приймати параметр scope? Як забезпечити, що ресурс (наприклад, база даних) відкривається один раз на всю тестову сесію?
6. Що таке conftest.py і в чому його перевага порівняно зі звичайним імпортом фікстур?
7. Поясніть роботу @pytest.mark.parametrize. Як уникнути дублювання тестового коду для функції, що тестується з 10 різними вхідними даними?
8. Що таке mock-об'єкт? Навіщо він потрібен? Як за допомогою @patch протестувати функцію, що робить HTTP-запит, без реального звернення до сервера?
9. Що таке покриття коду (code coverage)? Яка команда запускає pytest із вимірюванням покриття? Чому 100% покриття не гарантує відсутність помилок?
10. Поясніть цикл TDD: Red → Green → Refactor. Яка головна перевага написання тесту ДО реалізації коду?