**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 2. Python для прикладних задач

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

Перша частина цих методичних вказівок заклала міцний фундамент: ви навчилися мислити алгоритмічно, писати функції, будувати класи, тестувати код і обробляти помилки. Друга частина робить наступний крок – вона переносить отримані знання у площину реальних задач, з якими стикається сучасний фахівець. Тут Python постає не як навчальна мова, а як **інструмент для вирішення прикладних проблем**: від обробки великих масивів даних і побудови веб-сервісів до автоматизованого збору інформації з інтернету.

**Лабораторна робота 9** відкриває другу частину темою шаблонів проектування – перевірених архітектурних рішень для типових задач розробки. Знання Singleton, Factory, Observer, Strategy та інших патернів дозволяє проектувати код, який легко розширювати і змінювати. **Лабораторна робота 10** занурюється в асинхронне програмування та генератори: інструменти, що дозволяють ефективно обробляти великі обсяги даних і виконувати кілька задач конкурентно – без блокування основного потоку виконання.

**Лабораторні роботи 11 і 12** охоплюють два найпопулярніших напрямки практичної розробки. FastAPI – сучасний фреймворк для створення веб-API – демонструє, як за кілька годин побудувати повноцінний REST-сервіс з автоматичною документацією і валідацією даних. SQLAlchemy і SQLite показують, як Python-застосунки зберігають і отримують дані з реляційних баз: від прямих SQL-запитів до зручного ORM-підходу. **Лабораторна робота 13** присвячена автоматизованому збору даних: бібліотеки requests і BeautifulSoup для статичних сторінок і Selenium для динамічних – незамінний арсенал для моніторингу, досліджень і аналітики.

**Лабораторні роботи 14 і 15** завершують курс темою, що стала однією з головних рушійних сил популярності Python, – **аналізом даних**. NumPy і Matplotlib забезпечують числові обчислення та наукову візуалізацію, що широко використовуються у фізиці, математиці та інженерії. Pandas – центральний інструмент аналітика: завантаження таблиць, очищення «брудних» даних, групування, злиття і статистика. Ці навички є прямим шляхом до таких напрямків, як Data Science, Machine Learning та Business Intelligence.

Друга частина збірника відображає реальний стан індустрії: кожна тема відповідає конкретній професійній ролі або сфері знань. Бекенд-розробник будує API. Інженер даних автоматизує збір і обробку інформації. Аналітик досліджує датасети і будує графіки. Архітектор застосовує шаблони проектування. Усі вони пишуть Python. Завершивши цей курс, ви матимете практичний досвід у кожному з цих напрямків, достатній для того, щоб впевнено починати реальні проекти, продовжувати навчання у спеціалізованих курсах або виходити на ринок праці з конкретними, затребуваними навичками.

Лабораторна робота №9.

# Шаблони проектування в Python

**Мета роботи:** Зрозуміти поняття шаблону проектування як перевіреного рішення типової задачі; ознайомитися з класифікацією шаблонів та навчитися розпізнавати ситуації, де їх застосування є доцільним. Навчитися реалізовувати ключові породжувальні шаблони (Singleton, Factory Method, Builder) та структурні шаблони (Decorator, Facade, Proxy) засобами Python з використанням особливостей мови. Засвоїти поведінкові шаблони (Observer, Strategy, Iterator) і Python-специфічні підходи (Context Manager, Descriptor, Mixin); навчитися обирати шаблон, що відповідає конкретній задачі, і обґрунтовувати цей вибір.

## Теоретичні відомості:

Шаблони проектування (design patterns) – це перевірені, повторно використовувані рішення типових проблем, що виникають при проектуванні програмного забезпечення. Це не готовий код, а скоріше «рецепт» або «кресленик», який описує як організувати класи та об'єкти для вирішення конкретної задачі.
Концепцію шаблонів систематизували чотири автори – «банда чотирьох» – у книзі «Design Patterns» (1994). Вони описали 23 класичних шаблони, розбитих на три категорії:

- Породжувальні (Creational) – управляють процесом створення об'єктів
- Структурні (Structural) – описують способи компонування об'єктів у більші структури
- Поведінкові (Behavioral) – визначають алгоритми та обов'язки між об'єктами

Python як мова з розвинутими можливостями (декоратори, метакласи, протоколи) часто дозволяє реалізовувати шаблони значно лаконічніше ніж у Java чи C++. Деякі шаблони вже «вбудовані» у мову у вигляді синтаксичних конструкцій.

### *Коли застосовувати шаблони?*

*Шаблони – інструмент, а не самоціль. Не застосовуйте їх "про запас".*
*Застосовуйте шаблон коли ви вже розумієте проблему і бачите, що шаблон її вирішує.*
*Ознака доцільності: код без шаблону стає складним, дублюється або важко розширюється.*
*Ознака зловживання: додавання шаблону ускладнює простий код без реальної потреби.*

## Частина І. Породжувальні шаблони

**Singleton** – *Породжувальний*

*Проблема: Потрібно гарантувати, що клас має лише один екземпляр (конфігурація, підключення до БД, логер).*
Рішення: Зберігати єдиний екземпляр у змінній класу і повертати його при кожному зверненні.
Застосовувати коли: Клас представляє ресурс, який має існувати в одному екземплярі протягом усього часу роботи програми.

Переваги: Контрольований доступ до єдиного ресурсу; економія пам'яті; спільний стан.

Уявіть, що ваша програма має модуль конфігурації: файл читається один раз, але до налаштувань звертаються з різних частин коду. Якщо кожне звернення створює новий об'єкт – файл читатиметься знову і знову, і зміни не будуть спільними. Singleton вирішує це: перший виклик створює об'єкт, всі наступні – повертають той самий.

```
# — Реалізація через __new__ —
class Config:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance._initialized = False
        return cls._instance

    def __init__(self):
        if self._initialized:
            return
        self.debug = False
        self.db_url = 'sqlite:///app.db'
        self.max_connections = 10
        self._initialized = True

# Перевірка: обидва вказують на один об'єкт
c1 = Config()
c2 = Config()
print(c1 is c2)         # True
print(id(c1) == id(c2)) # True

c1.debug = True
print(c2.debug)         # True – той самий об'єкт!

# — Реалізація через декоратор (Pythonic підхід) —
def singleton(cls):
    instances = {}
    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]
    return get_instance

@singleton
class DatabaseConnection:
    def __init__(self, url: str):
        self.url = url
        self.connected = True
        print(f'Підключення до {url}')

db1 = DatabaseConnection('sqlite:///app.db')  # Підключення до...
db2 = DatabaseConnection('sqlite:///app.db')  # (без виводу!)
print(db1 is db2)  # True
```

**Factory Method** – *Породжувальний*

*Проблема: Код знає, що потрібно створити об'єкт певного типу, але точний тип визначається під час виконання.*

Рішення: Визначити інтерфейс створення об'єктів, але делегувати вибір конкретного класу підкласам або фабричній функції.

Застосовувати коли: Потрібно створювати об'єкти різних класів з однаковим інтерфейсом залежно від умов або параметрів.

Переваги: Заміна new() по всьому коду одним місцем; легко розширити новими типами без зміни клієнтського коду.

Уявіть програму для відправки повідомлень: залежно від налаштувань потрібно надіслати Email, SMS або Push-сповіщення. Без фабрики код пересипаний умовами if/elif скрізь. Фабричний метод зосереджує логіку вибору в одному місці, а клієнт просто просить «дай мені відправника» – не знаючи, який саме.

```python
from abc import ABC, abstractmethod

# ── Загальний інтерфейс продукту ──
class Notifier(ABC):
    @abstractmethod
    def send(self, recipient: str, message: str) -> None: ...

class EmailNotifier(Notifier):
    def send(self, recipient, message):
        print(f'Email → {recipient}: {message}')

class SMSNotifier(Notifier):
    def send(self, recipient, message):
        print(f'SMS → {recipient}: {message}')

class PushNotifier(Notifier):
    def send(self, recipient, message):
        print(f'Push → {recipient}: {message}')

# ── Фабрична функція (Pythonic стиль) ──
def create_notifier(channel: str) -> Notifier:
    registry = {
        'email': EmailNotifier,
        'sms':   SMSNotifier,
        'push':  PushNotifier,
    }
    cls = registry.get(channel.lower())
    if cls is None:
        raise ValueError(f'Невідомий канал: {channel}')
    return cls()

# Клієнт не знає про EmailNotifier / SMSNotifier
for channel in ['email', 'sms', 'push']:
    notifier = create_notifier(channel)
    notifier.send('user@example.com', 'Ваше замовлення готове')

# Додати новий канал – лише один рядок у registry!
```

**Builder** – *Породжувальний*
*Проблема: Об'єкт має багато необов'язкових параметрів і різних конфігурацій – конструктор стає перевантаженим.*
Рішення: Розбити процес побудови об'єкта на послідовні кроки через окремий клас-будівельник.
Застосовувати коли: Об'єкт потребує складної покрокової ініціалізації; бажано чіткий і читабельний код конструювання.
Переваги: Читабельний fluent API; легко варіювати конфігурації; відділяє конструювання від представлення.

Уявіть HTTP-запит: він може мати URL, метод, заголовки, тіло, таймаут, авторизацію – і більшість цих параметрів опціональні. Конструктор з 8 аргументами незручний. Builder дозволяє будувати запит крок за кроком у стилі «ланцюжка» (method chaining): `request.url(...).header(...).body(...).build()` – такий код читається як опис самого запиту.

```
class QueryBuilder:
    '''Будівельник SQL-запитів (спрощена версія).'''

    def __init__(self, table: str):
        self._table   = table
        self._fields  = ['*']
        self._where   = []
        self._order   = None
        self._limit   = None

    def select(self, *fields: str) -> 'QueryBuilder':
        self._fields = list(fields)
        return self          # повертаємо self для chaining

    def where(self, condition: str) -> 'QueryBuilder':
        self._where.append(condition)
        return self

    def order_by(self, field: str, desc: bool = False) -> 'QueryBuilder':
        direction = 'DESC' if desc else 'ASC'
        self._order = f'{field} {direction}'
        return self

    def limit(self, n: int) -> 'QueryBuilder':
        self._limit = n
        return self

    def build(self) -> str:
        query = f"SELECT {', '.join(self._fields)} FROM {self._table}"
        if self._where:
            query += ' WHERE ' + ' AND '.join(self._where)
        if self._order:
            query += f' ORDER BY {self._order}'
        if self._limit:
            query += f' LIMIT {self._limit}'
        return query

# Читабельний fluent API
query = (
```

```
    QueryBuilder('students')
    .select('name', 'gpa', 'year')
    .where('gpa >= 4.0')
    .where('year = 2')
    .order_by('gpa', desc=True)
    .limit(10)
    .build()
)
print(query)
# SELECT name, gpa, year FROM students
# WHERE gpa >= 4.0 AND year = 2 ORDER BY gpa DESC LIMIT 10
```

## Частина ІІ. Структурні шаблони

**Decorator** – *Структурний*

*Проблема: Потрібно динамічно додавати поведінку до об'єктів без зміни їхнього класу та без спадкування.*

Рішення: Обгорнути об'єкт у клас-обгортку (wrapper) з тим самим інтерфейсом, що розширює поведінку.

Застосовувати коли: Треба гнучко комбінувати різні функціональності; спадкування призвело б до вибуху кількості підкласів.

Переваги: Гнучкість: комбінації нескладних обгорток замість складної ієрархії; відповідність Open/Closed принципу.

Python має вбудовану синтаксичну підтримку декораторів через @. Але шаблон Decorator стосується не лише функцій – він описує обгортання об'єктів. Уявіть систему знижок у магазині: базова ціна, потім знижка на день народження, потім знижка для студентів. Замість одного складного класу – три прості обгортки, що комбінуються.

```
from abc import ABC, abstractmethod

# — Об'єктний Decorator (GoF) —
class Coffee(ABC):
    @abstractmethod
    def cost(self) -> float: ...
    @abstractmethod
    def description(self) -> str: ...

class SimpleCoffee(Coffee):
    def cost(self) -> float:        return 25.0
    def description(self) -> str:   return 'Кава'

class CoffeeDecorator(Coffee):
    def __init__(self, coffee: Coffee):
        self._coffee = coffee
    def cost(self) -> float:        return self._coffee.cost()
    def description(self) -> str:   return self._coffee.description()

class Milk(CoffeeDecorator):
    def cost(self):        return self._coffee.cost() + 8.0
    def description(self): return self._coffee.description() + ' + молоко'

class Sugar(CoffeeDecorator):
    def cost(self):        return self._coffee.cost() + 3.0
```

```
    def description(self): return self._coffee.description() + ' + цукор'

class Vanilla(CoffeeDecorator):
    def cost(self):        return self._coffee.cost() + 12.0
    def description(self): return self._coffee.description() + ' + ваніль'

# Комбінування обгорток
coffee = SimpleCoffee()
coffee = Milk(coffee)
coffee = Sugar(coffee)
coffee = Vanilla(coffee)

print(coffee.description())  # Кава + молоко + цукор + ваніль
print(f'{coffee.cost():.1f} грн')  # 48.0 грн

# ── Функціональний декоратор (Python-спосіб) ──
import functools, time

def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start  = time.perf_counter()
        result = func(*args, **kwargs)
        print(f'{func.__name__}: {time.perf_counter()-start:.4f}c')
        return result
    return wrapper

@timer
def heavy_computation(n):
    return sum(i**2 for i in range(n))

heavy_computation(1_000_000)  # heavy_computation: 0.1234c
```

**Facade** – *Структурний*

*Проблема: Підсистема складається з багатьох класів із складними залежностями; клієнт мусить знати деталі всіх.*

Рішення: Надати єдиний спрощений інтерфейс (фасад) для роботи зі складною підсистемою.

Застосовувати коли: Потрібно надати простий API для складної системи; клієнтський код не повинен знати деталей реалізації.

Переваги: Спрощення клієнтського коду; ізоляція клієнтів від складності; легша зміна підсистеми.

Фасад – це як "кнопка" на складному пристрої. Всередині кавомашини є насос, нагрівач, помельник, таймер – але ви просто натискаєте одну кнопку. Фасад ховає всі складнощі. У програмуванні це часто виглядає як клас-сервіс, що координує роботу кількох підсистем: завантаження файлу, валідація, збереження у БД, відправка email.

```
# Складна підсистема з кількох класів
class VideoDecoder:
    def decode(self, file: str) -> bytes:
        print(f'Декодуємо відео: {file}')
        return b'raw_video_data'

class AudioDecoder:
    def decode(self, file: str) -> bytes:
```

```
        print(f'Декодуємо аудіо: {file}')
        return b'raw_audio_data'

class VideoEncoder:
    def encode(self, data: bytes, fmt: str) -> bytes:
        print(f'Кодуємо відео у {fmt}')
        return b'encoded_video'

class FileWriter:
    def write(self, data: bytes, path: str) -> None:
        print(f'Зберігаємо у {path}')

# ── Фасад: простий інтерфейс для всієї підсистеми ──
class VideoConverterFacade:
    '''Конвертує відеофайли – клієнт не знає деталей.'''

    def __init__(self):
        self._video_dec = VideoDecoder()
        self._audio_dec = AudioDecoder()
        self._encoder   = VideoEncoder()
        self._writer    = FileWriter()

    def convert(self, source: str, output: str, fmt: str = 'mp4') -> None:
        video = self._video_dec.decode(source)
        audio = self._audio_dec.decode(source)
        encoded = self._encoder.encode(video + audio, fmt)
        self._writer.write(encoded, output)
        print(f'Готово: {output}')

# Клієнтський код: один виклик замість чотирьох класів
converter = VideoConverterFacade()
converter.convert('lecture.avi', 'lecture.mp4')
```

**Proxy** – *Структурний*

*Проблема: Потрібно контролювати доступ до об'єкта: кешувати результати, перевіряти права, відкладати ініціалізацію.*

Рішення: Створити об'єкт-замісник (proxy) з тим самим інтерфейсом, що перехоплює звернення до реального об'єкта.

Застосовувати коли: Дорога операція виконується повторно (кешуючий proxy); потрібна перевірка прав доступу; об'єкт важко ініціалізувати.

Переваги: Прозорість для клієнта; кешування; захист; відкладена ініціалізація.

Proxy – це посередник, що стоїть між клієнтом і реальним сервісом. Клієнт навіть не знає, що працює з proxy. Найчастіше використовується для кешування: база даних повільна, але якщо зберігати результат запиту в пам'яті – повторний виклик поверне відповідь миттєво. Proxy вирішує це прозоро для клієнта.

```
from abc import ABC, abstractmethod
import time

class DataService(ABC):
    @abstractmethod
    def fetch(self, query: str) -> dict: ...
```

```python
class RealDataService(DataService):
    '''Повільний сервіс (імітація запиту до БД).'''
    def fetch(self, query: str) -> dict:
        time.sleep(0.5)   # імітація повільної БД
        return {'query': query, 'result': 'data_' + query}

class CachingProxy(DataService):
    '''Proxy з кешуванням — той самий інтерфейс, але швидше.'''

    def __init__(self, service: DataService):
        self._service = service
        self._cache: dict = {}
        self._hits = 0

    def fetch(self, query: str) -> dict:
        if query in self._cache:
            self._hits += 1
            print(f'[Cache HIT] {query}')
            return self._cache[query]

        print(f'[Cache MISS] {query} — звертаємось до сервісу')
        result = self._service.fetch(query)
        self._cache[query] = result
        return result

    @property
    def cache_hits(self) -> int:
        return self._hits

# Клієнт не знає що працює з proxy
service = CachingProxy(RealDataService())

print(service.fetch('students'))   # Cache MISS — 0.5с
print(service.fetch('courses'))    # Cache MISS — 0.5с
print(service.fetch('students'))   # Cache HIT  — миттєво!
print(service.fetch('students'))   # Cache HIT  — миттєво!
print(f'Кеш-хітів: {service.cache_hits}')  # 2
```

## Частина III. Поведінкові шаблони

**Observer** – *Поведінковий*

*Проблема: Зміна стану одного об'єкта має автоматично відображатися на залежних об'єктах, але тісне зв'язування небажане.*

Рішення: Об'єкт-видавець (publisher) зберігає список підписників (subscribers) і сповіщає їх при зміні стану.

Застосовувати коли: Один об'єкт впливає на невизначену кількість інших; об'єкти мають реагувати на події без жорстких зв'язків.

Переваги: Слабке зв'язування; легко додавати/видаляти підписників; основа подієво-орієнтованих систем.

Спостерігач – основа будь-якої подієво-орієнтованої системи. Уявіть інтернет-магазин: коли товар з'являється у наявності – система має автоматично повідомити всіх клієнтів, що підписалися на сповіщення. Список клієнтів може змінюватися, і магазин не повинен про них знати нічого конкретного – лише те, що у них є метод notify(). Це і є Observer.

```python
from abc import ABC, abstractmethod
from typing import List

class Observer(ABC):
    @abstractmethod
    def update(self, event: str, data: dict) -> None: ...

class EventBus:
    '''Видавець подій – зберігає і сповіщає підписників.'''

    def __init__(self):
        self._subscribers: dict[str, List[Observer]] = {}

    def subscribe(self, event: str, observer: Observer) -> None:
        self._subscribers.setdefault(event, []).append(observer)

    def unsubscribe(self, event: str, observer: Observer) -> None:
        self._subscribers.get(event, []).remove(observer)

    def publish(self, event: str, data: dict = None) -> None:
        for obs in self._subscribers.get(event, []):
            obs.update(event, data or {})

# ── Конкретні спостерігачі ──
class EmailAlert(Observer):
    def __init__(self, email: str):
        self.email = email
    def update(self, event, data):
        print(f'Email [{self.email}]: {event} – {data}')

class SMSAlert(Observer):
    def __init__(self, phone: str):
        self.phone = phone
    def update(self, event, data):
        print(f'SMS [{self.phone}]: {event}')

class AuditLog(Observer):
    def update(self, event, data):
        print(f'[LOG] {event}: {data}')

# ── Використання ──
bus = EventBus()
email = EmailAlert('admin@store.com')
sms   = SMSAlert('+380671234567')
log   = AuditLog()

bus.subscribe('order.created', email)
bus.subscribe('order.created', log)
bus.subscribe('low.stock',     sms)
bus.subscribe('low.stock',     email)

bus.publish('order.created', {'order_id': 42, 'total': 1500})
bus.publish('low.stock',     {'product': 'Python Book', 'qty': 2})
```

**Strategy** – *Поведінковий*

*Проблема: Алгоритм може варіюватися залежно від умов; жорстке кодування if/elif для кожного варіанту ускладнює код.*

Рішення: Інкапсулювати кожен варіант алгоритму в окремий клас і дозволити замінювати їх незалежно.

Застосовувати коли: Є кілька схожих алгоритмів (сортування, оплата, шифрування); алгоритм потрібно змінювати динамічно.

Переваги: Легка заміна алгоритмів; усунення умовних операторів; новий алгоритм – новий клас без зміни існуючого.

Стратегія – це спосіб вибрати «як робити» незалежно від «що робити». Уявіть систему оплати: PayPal, картка, криптовалюта – логіка оплати різна, але результат однаковий. Замість великого if-elif у методі checkout(), кожен спосіб оплати стає окремою стратегією. Завтра додасьте Apple Pay – жодного рядка існуючого коду не зміниться.

```python
from abc import ABC, abstractmethod

class SortStrategy(ABC):
    @abstractmethod
    def sort(self, data: list) -> list: ...

class BubbleSort(SortStrategy):
    def sort(self, data: list) -> list:
        arr = data[:]
        for i in range(len(arr)):
            for j in range(len(arr)-i-1):
                if arr[j] > arr[j+1]:
                    arr[j], arr[j+1] = arr[j+1], arr[j]
        return arr

class QuickSort(SortStrategy):
    def sort(self, data: list) -> list:
        if len(data) <= 1: return data
        pivot = data[len(data)//2]
        left  = [x for x in data if x < pivot]
        mid   = [x for x in data if x == pivot]
        right = [x for x in data if x > pivot]
        return self.sort(left) + mid + self.sort(right)

class PythonSort(SortStrategy):
    def sort(self, data: list) -> list:
        return sorted(data)

# — Контекст: використовує стратегію —
class Sorter:
    def __init__(self, strategy: SortStrategy):
        self._strategy = strategy

    def set_strategy(self, strategy: SortStrategy) -> None:
        self._strategy = strategy

    def sort(self, data: list) -> list:
        return self._strategy.sort(data)

data = [5, 2, 8, 1, 9, 3, 7]
```

```
sorter = Sorter(BubbleSort())
print(sorter.sort(data))    # [1, 2, 3, 5, 7, 8, 9]

sorter.set_strategy(QuickSort())
print(sorter.sort(data))    # [1, 2, 3, 5, 7, 8, 9]

# Python-підхід: callable як стратегія
strategies = {
    'asc':    lambda d: sorted(d),
    'desc':   lambda d: sorted(d, reverse=True),
    'length': lambda d: sorted(d, key=len) if d else d,
}
apply = strategies.get('desc', strategies['asc'])
print(apply(data))          # [9, 8, 7, 5, 3, 2, 1]
```

## Частина IV. Python-специфічні підходи

Менеджер контексту – Python-специфічний шаблон управління ресурсами. Він гарантує виконання коду очищення (закриття файлу, відкат транзакції) незалежно від того, виникла помилка чи ні. По суті – це реалізація шаблону RAII (Resource Acquisition Is Initialization).
Менеджер контексту особливо корисний коли ресурс потребує парних дій «відкрити-закрити» або «увімкнути-вимкнути». Без нього легко забути закрити файл або звільнити з'єднання. З ним – це гарантовано відбудеться навіть при винятку.

```
from contextlib import contextmanager
import time

# — Через клас (__enter__ / __exit__) —
class Timer:
    '''Вимірює час виконання блоку коду.'''

    def __enter__(self):
        self._start = time.perf_counter()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.elapsed = time.perf_counter() - self._start
        print(f'Час виконання: {self.elapsed:.4f}c')
        return False   # не пригнічуємо винятку

with Timer() as t:
    result = sum(i**2 for i in range(1_000_000))
print(f'elapsed: {t.elapsed:.4f}c')

# — Через @contextmanager (простіший спосіб) —
@contextmanager
def transaction(db):
    '''Менеджер транзакцій: commit або rollback.'''
    print('BEGIN TRANSACTION')
    try:
        yield db
        print('COMMIT')
        db['committed'] = True
    except Exception as e:
        print(f'ROLLBACK: {e}')
        db['committed'] = False
```

```
        raise

db = {}
with transaction(db) as conn:
    conn['data'] = 'важливі дані'
print(db)  # {'data': 'важливі дані', 'committed': True}
```

Mixin – це клас, що надає методи для використання іншими класами через множинне успадкування, але сам не призначений для безпосереднього інстанціювання. Mixins вирішують проблему горизонтального повторного використання коду: одна і та ж поведінка (наприклад, серіалізація у JSON) потрібна у декількох непов'язаних класах.
На відміну від звичайного успадкування, Mixin додає «бічну» функціональність: клас «є» якоюсь сутністю, але також «вміє» щось – завдяки домішці. Наприклад, Student «є» Person і «вміє» серіалізуватися у JSON та порівнюватися.

```
import json

class JSONMixin:
    '''Домішка: серіалізація у JSON.'''
    def to_json(self) -> str:
        return json.dumps(self.__dict__, ensure_ascii=False, indent=2)

    @classmethod
    def from_json(cls, json_str: str):
        data = json.loads(json_str)
        obj  = cls.__new__(cls)
        obj.__dict__.update(data)
        return obj

class ComparableMixin:
    '''Домішка: порівняння за _compare_key.'''
    def _compare_key(self):
        raise NotImplementedError
    def __eq__(self, other): return self._compare_key() == other._compare_key()
    def __lt__(self, other): return self._compare_key() <  other._compare_key()
    def __le__(self, other): return self._compare_key() <= other._compare_key()

class ReprMixin:
    '''Домішка: автоматичний __repr__.'''
    def __repr__(self) -> str:
        attrs = ', '.join(f'{k}={v!r}' for k, v in self.__dict__.items())
        return f'{self.__class__.__name__}({attrs})'

# ── Основний клас із домішками ──
class Student(JSONMixin, ComparableMixin, ReprMixin):
    def __init__(self, name: str, gpa: float):
        self.name = name
        self.gpa  = gpa
    def _compare_key(self): return self.gpa

s1 = Student('Іван', 4.2)
s2 = Student('Марія', 4.8)

print(repr(s1))                    # Student(name='Іван', gpa=4.2)
print(s1 < s2)                     # True
print(sorted([s2, s1], reverse=True))  # Марія, Іван
```

```python
json_str = s1.to_json()
s3 = Student.from_json(json_str)
print(s1 == s3)                    # True
```

## Завдання на лабораторну роботу

Для кожного завдання напишіть окремий файл. Кожна реалізація шаблону має супроводжуватися демонстраційним кодом, що чітко показує поведінку шаблону.

9.1. Singleton та Factory Method.Реалізуйте два шаблони у контексті системи логування:

- Singleton: клас AppLogger – єдиний логер застосунку. При першому створенні відкриває файл app.log, при наступних – повертає той самий екземпляр. Метод log(level, message) дописує у файл і виводить у консоль;
- Factory Method: функція create_formatter(fmt_type) повертає один із форматів: 'plain' – простий текст, 'csv' – рядок через кому, 'json' – JSON-рядок. Кожен формат реалізується окремим класом з методом format(level, message, timestamp).

Продемонструйте: логер є єдиним (перевірте через is); форматери взаємозамінні без зміни коду логера.

9.2. Builder: конструктор HTML-елементів. Реалізуйте клас HTMLBuilder для побудови HTML-рядків у стилі fluent API:

- tag(name) – встановити тег (div, p, span, button тощо);
- id(value), css_class(*classes), style(**props) – атрибути;
- text(content) – текстовий вміст;
- child(builder) – вкладений елемент (приймає інший HTMLBuilder);
- build() → str – збирає рядок HTML.

Побудуйте картку студента: div.card з заголовком h2 (ім'я), параграфом (GPA), кнопкою 'Деталі'. Порівняйте читабельність із ручним рядковим конкатенуванням.

9.3. Decorator та Context Manager.

Реалізуйте функціональні декоратори (через @functools.wraps):

- @retry(times=3, delay=0.1, exceptions=(Exception,)) – повторює виклик до times разів при виникненні вказаних винятків;
- @validate_args(**validators) – перевіряє аргументи функції: validate_args(x=lambda v: v>0, y=lambda v: isinstance(v,str)) кидає ValueError якщо перевірка не пройдена.

Реалізуйте контекстний менеджер @contextmanager indent_print(level=1) – всі print() всередині блоку мають виводитися з відступом level * 2 пробілів. (Підказка: підмініть sys.stdout на клас-обгортку.)

9.4. Observer: система подій. Реалізуйте систему подій для університетського порталу:

- EventBus із методами subscribe(event, observer), unsubscribe, publish(event, data);
- Спостерігачі: EmailNotifier (виводить лист), SMSNotifier (виводить SMS), AuditLogger (записує у файл audit.log), Dashboard (зберігає статистику подій у словнику);
- Події: 'student.enrolled' (студент зарахований), 'grade.added' (виставлена оцінка), 'student.graduated' (закінчив навчання).

Продемонструйте: підписка кількох спостерігачів на одну подію; відписка; публікація мінімум 5 подій; вивід статистики Dashboard (кількість кожного типу події).

## Контрольні запитання

1. Що таке шаблон проектування? Чому не варто застосовувати їх «про запас»? Наведіть ознаки доцільного та недоцільного застосування.
2. На які три категорії поділяються GoF-шаблони? Наведіть по одному прикладу з кожної категорії.
3. Поясніть шаблон Singleton. Де він може спричинити проблеми (наприклад, у тестуванні)? Як реалізувати Singleton через декоратор у Python?
4. Чим Factory Method відрізняється від прямого виклику конструктора? Яку проблему він вирішує при додаванні нових типів?
5. Поясніть шаблон Builder. Що таке fluent API (method chaining)? Коли Builder є кращим вибором ніж конструктор з параметрами?
6. У чому різниця між об'єктним Decorator (GoF) і функціональним декоратором Python (@decorator)? Яку спільну ідею вони реалізують?
7. Поясніть шаблон Observer на прикладі. Що означає «слабке зв'язування» у цьому контексті? Чому видавець не знає деталей про підписників?
8. Поясніть шаблон Strategy. Яку альтернативу він пропонує великим блокам if-elif? Як Strategy використовує принцип Open/Closed?
9. Що таке Mixin? Чим він відрізняється від звичайного успадкування? Назвіть три обмеження при проектуванні Mixin-класів.
10. Що таке Context Manager у Python? Якому GoF-шаблону він найближчий і чому? Яку перевагу дає @contextmanager порівняно з реалізацією через клас?

Лабораторна робота №10.

**Генератори, ітератори та асинхронне програмування**

**Мета роботи:** Зрозуміти протокол ітерації Python – як реалізуються ітератори вручну через __iter__ та __next__, навчитися будувати ліниві послідовності, що не завантажують усі дані у пам'ять одночасно, і зрозуміти, чому генератори є ефективнішими за списки для великих об'ємів даних. Опанувати синтаксис функцій-генераторів (yield, yield from) та виразів-генераторів, навчитися будувати пайплайни обробки даних; ознайомитися з модулем itertools та його функціями для роботи з ітерованими послідовностями. Зрозуміти концепцію конкурентного виконання та asyncio – подієвий цикл, корутини, async/await, asyncio.gather() і asyncio.create_task(); навчитися писати асинхронні програми для операцій введення-виведення та мережевих запитів.

**Теоретичні відомості:**

Будь-який об'єкт, що підтримує цикл `for`, реалізує протокол ітерації. Він складається з двох методів:

- `__iter__(self)` – повертає сам ітератор (або інший об'єкт-ітератор);
- `__next__(self)` – повертає наступний елемент або кидає StopIteration коли елементи закінчились.

Різниця між ітерованим об'єктом (iterable) та ітератором (iterator):

- Iterable – об'єкт, з якого можна отримати ітератор: `list, str, dict, range` – вони мають `__iter__`;
- Iterator – об'єкт, що безпосередньо перебирає елементи: має і `__iter__`, і `__next__`.

```python
# ── Як Python виконує цикл for ──
lst = [10, 20, 30]

# for x in lst: print(x)
# Еквівалентно до:
iterator = iter(lst)          # виклик lst.__iter__()
while True:
    try:
        x = next(iterator)    # виклик iterator.__next__()
        print(x)
    except StopIteration:
        break                 # ітерація завершена

# ── Перевірка протоколу ──
lst = [1, 2, 3]
it  = iter(lst)
print(next(it))   # 1
print(next(it))   # 2
print(next(it))   # 3
# next(it)        # StopIteration!

# Список – iterable, але НЕ iterator
print(hasattr(lst, '__iter__'))   # True
print(hasattr(lst, '__next__'))   # False

# Ітератор – і iterable, і iterator
print(hasattr(it, '__iter__'))    # True
```

```
print(hasattr(it, '__next__'))    # True
print(iter(it) is it)             # True  – iter(iterator) повертає себе
```

Реалізація власного ітератора

```
class Fibonacci:
    '''Ітератор послідовності Фібоначчі до заданої межі.'''

    def __init__(self, limit: int):
        self.limit = limit
        self.a, self.b = 0, 1

    def __iter__(self):
        return self          # повертаємо себе

    def __next__(self):
        if self.a > self.limit:
            raise StopIteration
        value = self.a
        self.a, self.b = self.b, self.a + self.b
        return value


fib = Fibonacci(100)
print(list(fib))   # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

# Можна використовувати у циклі
for n in Fibonacci(50):
    print(n, end=' ')   # 0 1 1 2 3 5 8 13 21 34

# ── Клас зі складнішим станом ──
class Range:
    '''Власна реалізація range().'''

    def __init__(self, start: int, stop: int, step: int = 1):
        if step == 0:
            raise ValueError('step не може бути 0')
        self.start = start
        self.stop  = stop
        self.step  = step
        self._current = start

    def __iter__(self):
        self._current = self.start   # reset для повторного використання
        return self

    def __next__(self):
        if (self.step > 0 and self._current >= self.stop) or \
           (self.step < 0 and self._current <= self.stop):
            raise StopIteration
        value = self._current
        self._current += self.step
        return value

    def __repr__(self):
        return f'Range({self.start}, {self.stop}, {self.step})'


print(list(Range(1, 10, 2)))    # [1, 3, 5, 7, 9]
print(list(Range(10, 0, -3)))   # [10, 7, 4, 1]
```

Генератор – це функція, що містить `yield`. Замість return, що повертає значення і завершує функцію, yield повертає значення і призупиняє виконання, зберігаючи весь внутрішній стан. При наступному виклику next() виконання продовжується з того ж місця.
Генератор автоматично реалізує протокол ітерації – __iter__ та __next__ генеруються Python автоматично.

```
# ── Порівняння: функція vs генератор ──

# Звичайна функція: повертає ВЕСЬ список одразу
def squares_list(n):
    result = []
    for i in range(n):
        result.append(i ** 2)
    return result   # займає O(n) пам'яті

# Генератор: видає по одному значенню
def squares_gen(n):
    for i in range(n):
        yield i ** 2   # призупиняє і видає значення
    # StopIteration кидається автоматично після виходу з функції

# Список одразу займає пам'ять; генератор – майже нічого
import sys
lst = squares_list(1000)
gen = squares_gen(1000)
print(sys.getsizeof(lst))   # ~8856 байт
print(sys.getsizeof(gen))   # 112 байт

# Використання генератора
g = squares_gen(5)
print(next(g))   # 0
print(next(g))   # 1
print(next(g))   # 4
print(list(g))   # [9, 16]  - решта

# У циклі for
for sq in squares_gen(6):
    print(sq, end=' ')   # 0 1 4 9 16 25

# ── Генератор з кількома yield ──
def traffic_light():
    while True:         # нескінченний генератор
        yield 'Зелений'
        yield 'Жовтий'
        yield 'Червоний'

light = traffic_light()
for _ in range(7):
    print(next(light), end=' ')
# Зелений Жовтий Червоний Зелений Жовтий Червоний Зелений

# ── Генератор Фібоначчі (значно простіший за клас!) ──
def fibonacci(limit=None):
    a, b = 0, 1
    while limit is None or a <= limit:
        yield a
        a, b = b, a + b

print(list(fibonacci(100)))
```

```
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

# Нескінченний – беремо перші 10
from itertools import islice
print(list(islice(fibonacci(), 10)))
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Двобічний зв'язок: `send()` та `throw()`

```
# yield може ОТРИМУВАТИ значення через send()

def accumulator():
    '''Генератор-акумулятор: накопичує значення через send().'''
    total = 0
    while True:
        value = yield total   # yield повертає total, отримує value
        if value is None:
            break
        total += value

acc = accumulator()
next(acc)           # обов'язково просуваємо до першого yield
print(acc.send(10))   # 10
print(acc.send(20))   # 30
print(acc.send(5))    # 35

# ── throw(): передати виняток у генератор ──
def safe_gen():
    try:
        while True:
            yield 'значення'
    except ValueError as e:
        yield f'Перехоплено: {e}'

g = safe_gen()
print(next(g))                      # 'значення'
print(g.throw(ValueError, 'стоп')) # 'Перехоплено: стоп'

# ── close(): завершити генератор (кидає GeneratorExit) ──
def counting():
    try:
        i = 0
        while True:
            yield i
            i += 1
    finally:
        print('Генератор закрито')  # виконується при close()

g = counting()
print(next(g), next(g), next(g))   # 0 1 2
g.close()                           # 'Генератор закрито'
```

`yield from` делегує ітерацію до вкладеного ітерованого об'єкта або генератора. Це спрощує код і дозволяє будувати пайплайни з ланцюжків генераторів

```
# ── Порівняння: ручний цикл vs yield from ──

# Без yield from
def chain_manual(*iterables):
```

```
    for it in iterables:
        for item in it:
            yield item

# 3 yield from – коротше і ефективніше
def chain_yf(*iterables):
    for it in iterables:
        yield from it

print(list(chain_yf([1,2], 'ab', (3,4))))
# [1, 2, 'a', 'b', 3, 4]

# ── Сплощення вкладеної структури ──
def flatten(nested):
    for item in nested:
        if isinstance(item, (list, tuple)):
            yield from flatten(item)   # рекурсивне делегування
        else:
            yield item

data = [1, [2, 3, [4, 5]], 6, [7, [8, [9]]]]
print(list(flatten(data)))   # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# ── Пайплайн генераторів ──
# Кожен генератор обробляє потік даних «лінево»

def read_numbers(filename):
    '''Читає числа з файлу по одному.'''
    with open(filename) as f:
        for line in f:
            yield int(line.strip())

def filter_positive(numbers):
    '''Фільтрує тільки додатні.'''
    for n in numbers:
        if n > 0:
            yield n

def square(numbers):
    '''Підносить до квадрату.'''
    for n in numbers:
        yield n ** 2

def take(n, iterable):
    '''Бере перші n елементів.'''
    for i, item in enumerate(iterable):
        if i >= n: break
        yield item

# Пайплайн: файл → додатні → квадрати → перші 5
# Жодна проміжна колекція НЕ створюється у пам'яті!
# pipeline = take(5, square(filter_positive(read_numbers('data.txt'))))
# print(list(pipeline))

# ── yield from для передачі значень ──
def sub_gen():
    received = yield 'від підгенератора'
    yield f'отримано: {received}'
```

```python
def main_gen():
    result = yield from sub_gen()  # передає send() у sub_gen

g = main_gen()
print(next(g))          # 'від підгенератора'
print(g.send('привіт')) # 'отримано: привіт'
```

Вираз-генератор – це компактний синтаксис без функції: (вираз for змінна in ітероване if умова). На відміну від спискового включення у квадратних дужках, вираз у круглих дужках не створює список – він повертає генератор.

```python
import sys

# Списокове включення – одразу список у пам'яті
lst = [x ** 2 for x in range(10_000)]
print(sys.getsizeof(lst))    # ~87624 байт

# Вираз-генератор – ліниве обчислення
gen = (x ** 2 for x in range(10_000))
print(sys.getsizeof(gen))    # 112 байт

# ── Вбудований у виклик функції ──
# Одинарні дужки не потрібні коли генератор – єдиний аргумент
total = sum(x**2 for x in range(100))
max_v = max(len(w) for w in ['apple', 'banana', 'kiwi'])
any_e = any(x > 5 for x in [1, 3, 7, 2])
all_p = all(x > 0 for x in [1, 2, 3, 4])

print(total)   # 328350
print(max_v)   # 6
print(any_e)   # True
print(all_p)   # True

# ── З умовою ──
evens = (x for x in range(20) if x % 2 == 0)
print(list(evens))   # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

# ── Ланцюжок виразів-генераторів ──
words = ['hello', 'world', 'python', 'is', 'great']
result = list(
    w.upper() for w in words if len(w) > 4
)
print(result)   # ['HELLO', 'WORLD', 'PYTHON', 'GREAT']

# ── Коли що обрати? ──
# list comp:  потрібен список для індексування, багатократного обходу
# gen expr:   потрібен лише один прохід, або великий обсяг даних
# gen func:   складна логіка зі станом, кілька yield, yield from
```

Модуль itertools – стандартна бібліотека для ітераторів

| Концепція / Синтаксис | Опис / Результат |
|---|---|
| `count(start, step)` | Нескінченний лічильник: count(10, 2) → 10,12,14,... |
| `cycle(iterable)` | Нескінченне циклічне повторення: cycle('AB') → A,B,A,B,... |
| `repeat(obj, n)` | Повторює obj рівно n разів (або нескінченно) |
| `chain(*iters)` | Ланцюжок ітерованих: chain([1,2],[3,4]) → 1,2,3,4 |
| `islice(it, stop)` | Зріз ітератора: islice(count(), 5) → 0,1,2,3,4 |
| `takewhile(pred, it)` | Бере елементи поки pred(x) == True |

| | |
|---|---|
| `dropwhile(pred, it)` | Пропускає поки pred(x) == True, далі бере все |
| `filterfalse(pred, it)` | Елементи де pred(x) == False |
| `compress(data, sel)` | Фільтр за маскою: compress('ABCD',[1,0,1,0]) → A,C |
| `starmap(fn, pairs)` | map із розпакуванням: starmap(pow,[(2,3),(3,2)]) → 8,9 |
| `groupby(it, key)` | Групує за ключем (дані мають бути відсортовані!) |
| `product(*its)` | Декартовий добуток: product('AB','12') → A1,A2,B1,B2 |
| `permutations(it, r)` | Перестановки довжини r |
| `combinations(it, r)` | Комбінації без повторень довжини r |
| `accumulate(it, fn)` | Накопичення: accumulate([1,2,3]) → 1,3,6 |

```python
from itertools import (
    count, cycle, chain, islice, takewhile, dropwhile,
    groupby, product, combinations, accumulate, starmap
)

# count + islice: генеруємо перші 5 непарних
odd = islice(filter(lambda x: x % 2, count(1)), 5)
print(list(odd))   # [1, 3, 5, 7, 9]

# takewhile / dropwhile
data = [2, 4, 6, 7, 8, 9, 10]
print(list(takewhile(lambda x: x % 2 == 0, data)))  # [2, 4, 6]
print(list(dropwhile(lambda x: x % 2 == 0, data)))  # [7, 8, 9, 10]

# accumulate: поточна сума
sales = [100, 250, 180, 320, 90]
cumulative = list(accumulate(sales))
print(cumulative)   # [100, 350, 530, 850, 940]

# accumulate з функцією: поточний максимум
import operator
running_max = list(accumulate(sales, max))
print(running_max)  # [100, 250, 250, 320, 320]

# groupby: групуємо за першою літерою
words = sorted(['apple','ant','banana','bear','cherry','cat'])
for key, group in groupby(words, key=lambda w: w[0]):
    print(f'{key}: {list(group)}')
# a: ['ant', 'apple']
# b: ['banana', 'bear']
# c: ['cat', 'cherry']

# product: таблиця множення
for i, j in product(range(1, 4), repeat=2):
    print(f'{i}*{j}={i*j}', end='  ')

# combinations: пари студентів для проєкту
students = ['Іван', 'Марія', 'Олег', 'Ліна']
pairs = list(combinations(students, 2))
print(f'Можливих пар: {len(pairs)}')
for pair in pairs:
    print(pair)
```

## Асинхронне програмування: asyncio

Синхронний код виконується послідовно: кожна операція чекає завершення попередньої. Якщо програма виконує 10 мережевих запитів по 1 секунді кожен – вона чекає 10 секунд. Асинхронний код дозволяє «перемикатись» між задачами під час очікування. Поки один запит чекає відповіді сервера, інша задача може виконуватись – і загальний час скорочується до ~1 секунди.

- Конкурентність (concurrency) – кілька задач виконуються по черзі, перемикаючись між собою. Це asyncio.
- Паралелізм (parallelism) – задачі виконуються одночасно на різних ядрах процесора. Це multiprocessing.
- asyncio підходить для I/O-bound задач: мережа, файли, бази даних.
- multiprocessing підходить для CPU-bound задач: обчислення, кодування, стиснення.

| **Концепція / Синтаксис** | **Опис / Результат** |
|---|---|
| `async def func():` | Визначає корутину (coroutine) – асинхронну функцію |
| `await expr` | Призупиняє корутину поки expr не завершиться |
| `asyncio.run(coro)` | Запускає корутину: створює і запускає подієвий цикл |
| `asyncio.gather(*coros)` | Запускає кілька корутин паралельно, чекає всіх |
| `asyncio.create_task(c)` | Запускає корутину як незалежну задачу (Task) |
| `asyncio.sleep(n)` | Асинхронна затримка: поступається керуванням (не блокує!) |
| `asyncio.wait_for(c,t)` | Запускає корутину з таймаутом (asyncio.TimeoutError) |
| `asyncio.Queue()` | Асинхронна черга для обміну даними між задачами |

Основи async/await. Корутини

```python
import asyncio
import time

# ── Синхронний підхід (ПОВІЛЬНО) ──
def sync_task(name, delay):
    print(f'{name}: починаю')
    time.sleep(delay)      # БЛОКУЄ весь потік!
    print(f'{name}: завершено')

start = time.perf_counter()
sync_task('Задача 1', 2)
sync_task('Задача 2', 1)
sync_task('Задача 3', 1.5)
print(f'Синхронно: {time.perf_counter()-start:.2f}с')  # ~4.5с

# ── Асинхронний підхід (ШВИДКО) ──
async def async_task(name: str, delay: float) -> str:
    print(f'{name}: починаю')
    await asyncio.sleep(delay)  # ПОСТУПАЄТЬСЯ керуванням іншим задачам
    print(f'{name}: завершено')
    return f'{name} виконано за {delay}с'

async def main():
    start = time.perf_counter()

    # gather запускає всі корутини ОДНОЧАСНО
    results = await asyncio.gather(
```

```
        async_task('Задача 1', 2),
        async_task('Задача 2', 1),
        async_task('Задача 3', 1.5),
    )

    elapsed = time.perf_counter() - start
    print(f'Асинхронно: {elapsed:.2f}с')  # ~2.0с (найдовша задача)
    print(results)

asyncio.run(main())
# Задача 1: починаю
# Задача 2: починаю
# Задача 3: починаю
# Задача 2: завершено     (через 1с)
# Задача 3: завершено     (через 1.5с)
# Задача 1: завершено     (через 2с)
# Асинхронно: 2.00с
```

asyncio.Task та управління задачами

```
import asyncio
import time

# ── create_task: запуск без очікування ──
async def worker(name: str, delay: float):
    await asyncio.sleep(delay)
    return f'{name} завершено'

async def main():
    # create_task запускає задачу ОДРАЗУ у фоні
    task1 = asyncio.create_task(worker('A', 2), name='worker-A')
    task2 = asyncio.create_task(worker('B', 1), name='worker-B')
    task3 = asyncio.create_task(worker('C', 3), name='worker-C')

    print(f'Задачі запущені: {task1.get_name()}, {task2.get_name()}')

    # await чекає конкретну задачу
    result_b = await task2
    print(f'Задача B: {result_b}')

    # Скасування задачі
    task3.cancel()
    try:
        await task3
    except asyncio.CancelledError:
        print('Задача C скасована')

    print(f'Задача A: {await task1}')

asyncio.run(main())

# ── asyncio.wait: чекати першу або всі ──
async def with_wait():
    tasks = [
        asyncio.create_task(worker(f'T{i}', i*0.5))
        for i in range(1, 5)
    ]

    # Чекати першу виконану
```

```
    done, pending = await asyncio.wait(
        tasks, return_when=asyncio.FIRST_COMPLETED
    )
    print(f'Перша завершена: {done.pop().result()}')

    # Скасувати решту
    for t in pending:
        t.cancel()

asyncio.run(with_wait())

# ── wait_for: таймаут ──
async def slow_operation():
    await asyncio.sleep(5)
    return 'готово'

async def with_timeout():
    try:
        result = await asyncio.wait_for(slow_operation(), timeout=2.0)
    except asyncio.TimeoutError:
        print('Операція перевищила таймаут!')

asyncio.run(with_timeout())
```

Асинхронні черги. Паттерн Producer-Consumer

```
import asyncio
import random

# asyncio.Queue – потокобезпечна черга для передачі даних
# між корутинами

async def producer(queue: asyncio.Queue, n: int):
    '''Генерує задачі і кладе у чергу.'''
    for i in range(n):
        delay = random.uniform(0.1, 0.5)
        item = f'задача-{i+1}'
        await queue.put(item)
        print(f'[Producer] додав {item}')
        await asyncio.sleep(delay)
    # Сигнал завершення для кожного споживача
    await queue.put(None)
    await queue.put(None)

async def consumer(name: str, queue: asyncio.Queue):
    '''Бере задачі з черги і обробляє.'''
    while True:
        item = await queue.get()
        if item is None:          # сигнал зупинки
            queue.task_done()
            break
        work_time = random.uniform(0.2, 0.8)
        await asyncio.sleep(work_time)   # імітація обробки
        print(f'[{name}] оброблено {item} за {work_time:.2f}с')
        queue.task_done()

async def main():
    queue = asyncio.Queue(maxsize=5)  # черга з обмеженням
```

```
    # Запускаємо 1 виробника і 2 споживачі
    await asyncio.gather(
        producer(queue, 8),
        consumer('Споживач-1', queue),
        consumer('Споживач-2', queue),
    )
    print('Всі задачі виконано')

asyncio.run(main())
```

## Асинхронні генератори та контекстні менеджери

Python підтримує async for та async with – асинхронні аналоги звичайних конструкцій. Асинхронний генератор поєднує yield та await.

```
import asyncio

# ── Асинхронний генератор ──
async def async_range(start: int, stop: int, delay: float = 0.1):
    '''Генерує числа з асинхронною затримкою між ними.'''
    for i in range(start, stop):
        await asyncio.sleep(delay)
        yield i

async def use_async_gen():
    async for num in async_range(0, 5):
        print(num, end=' ')   # 0 1 2 3 4

    # Збирання у список
    result = [n async for n in async_range(0, 5)]
    print(result)  # [0, 1, 2, 3, 4]

asyncio.run(use_async_gen())

# ── Асинхронний контекстний менеджер ──
class AsyncDBConnection:
    '''Імітує асинхронне підключення до бази даних.'''

    def __init__(self, url: str):
        self.url = url
        self._connected = False

    async def __aenter__(self):
        print(f'Підключення до {self.url}...')
        await asyncio.sleep(0.1)   # імітація підключення
        self._connected = True
        print('Підключено!')
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        await asyncio.sleep(0.05)  # імітація закриття
        self._connected = False
        print('З\'єднання закрито')
        return False

    async def query(self, sql: str):
        if not self._connected:
            raise RuntimeError('Не підключено!')
```

```
        await asyncio.sleep(0.1)
        return f'Результат для: {sql}'

async def main():
    async with AsyncDBConnection('postgresql://localhost/mydb') as db:
        result = await db.query('SELECT * FROM users')
        print(result)

asyncio.run(main())

# — asynccontextmanager: через декоратор —
from contextlib import asynccontextmanager

@asynccontextmanager
async def managed_resource(name: str):
    print(f'Відкриваємо: {name}')
    await asyncio.sleep(0.1)
    try:
        yield name
    finally:
        await asyncio.sleep(0.05)
        print(f'Закриваємо: {name}')

async def demo():
    async with managed_resource('сесія') as res:
        print(f'Використовуємо {res}')

asyncio.run(demo())
```

Найпоширеніший реальний сценарій для asyncio – паралельна завантаження даних з кількох URL. Без асинхронності 10 запитів по 0.5 секунди займуть 5 секунд; з asyncio – лише ~0.5 секунди.

```
import asyncio
import time

# — Імітація без реальних запитів —
# Для реальних HTTP запитів: pip install aiohttp

async def fetch_url(url: str, delay: float) -> dict:
    '''Імітує HTTP-запит із затримкою.'''
    await asyncio.sleep(delay)   # імітація мережевої затримки
    return {'url': url, 'status': 200, 'size': len(url) * 100}

async def fetch_all(urls: list[str]) -> list[dict]:
    '''Завантажує всі URL паралельно.'''
    tasks = [
        asyncio.create_task(fetch_url(url, 0.5))
        for url in urls
    ]
    results = await asyncio.gather(*tasks, return_exceptions=True)
    return results

urls = [
    'https://api.example.com/users',
    'https://api.example.com/posts',
    'https://api.example.com/comments',
    'https://api.example.com/albums',
    'https://api.example.com/photos',
```

```
]

async def main():
    start = time.perf_counter()
    results = await fetch_all(urls)
    elapsed = time.perf_counter() - start

    for r in results:
        if isinstance(r, Exception):
            print(f'Помилка: {r}')
        else:
            print(f'OK: {r["url"]} ({r["size"]} байт)')

    print(f'Завантажено {len(urls)} URL за {elapsed:.2f}с')
    print(f'(синхронно зайняло б ~{len(urls)*0.5:.1f}с)')

asyncio.run(main())

# ── З реальним aiohttp (для довідки) ──
# import aiohttp
#
# async def real_fetch(session, url):
#     async with session.get(url) as response:
#         return await response.json()
#
# async def real_fetch_all(urls):
#     async with aiohttp.ClientSession() as session:
#         tasks = [real_fetch(session, url) for url in urls]
#         return await asyncio.gather(*tasks)
```

## Завдання на лабораторну роботу

Для кожного завдання створіть окремий файл task1.py … task7.py. Для завдань з asyncio обов'язково використовуйте `asyncio.run()` як точку входу.

10.1. Реалізуйте клас-ітератор Primes(limit), що генерує всі прості числа до заданої межі:

- реалізуйте __iter__ та __next__ вручну (без використання yield);
- в __next__ перевіряйте кожне число методом is_prime() (приватний статичний метод класу);
- клас повинен підтримувати повторну ітерацію (after iter() reset);
- додайте __len__ що повертає кількість простих чисел до limit (обчисліть при init);
- продемонструйте: list(Primes(50)), for p in Primes(30), 17 in Primes(20).

Також реалізуйте генераторну версію gen_primes(limit) та порівняйте споживання пам'яті через sys.getsizeof.

10.2. Реалізуйте такі генераторні функції та вирази:

- countdown(n) – зворотній відлік від n до 0;
- running_average(iterable) – генератор поточного середнього: для [4, 8, 2, 6] дає 4.0, 6.0, 4.67, 5.0;
- window(iterable, size) – ковзне вікно: window([1,2,3,4,5], 3) → (1,2,3), (2,3,4), (3,4,5);
- flatten(nested) – рекурсивне сплощення списку довільної глибини через yield from.

Для кожної функції також напишіть еквівалентний вираз-генератор (де це можливо) і

порівняйте обидва варіанти. Продемонструйте роботу з великими даними (range(1_000_000)) і виміряйте пам'ять.

10.3. Вирішіть наступні задачі, використовуючи виключно функції з itertools (без явних циклів де можливо):

- Згенеруйте нескінченну послідовність чисел виду 1, -1, 2, -2, 3, -3, ... і візьміть перші 12 значень (islice + count + cycle або власна комбінація);
- Знайдіть усі комбінації по 3 студенти з групи із 6 для проєктних команд; підрахуйте кількість комбінацій
- Для списку щомісячних продажів [120,95,180,210,145,90,220,185,160,200,175,140] знайдіть поточний максимум та мінімум через accumulate
- Згрупуйте список слів за довжиною через groupby (попередньо відсортуйте)
- Знайдіть усі 4-літерні паролі з символів 'abc123' через product

10.4. Побудуйте ланцюжок генераторів для обробки «великого» текстового файлу (або великого рядка). Жодна проміжна колекція не повинна зберігатись у пам'яті цілком. Ланцюжок обробки:

- read_lines(filename) або generate_lines(text) → рядки файлу
- parse_words(lines) → окремі слова (split + strip розділових знаків)
- normalize(words) → слова у нижньому регістрі, без пустих
- filter_stopwords(words, stopwords) → без стоп-слів
- take_n(n, iterable) → перші n елементів

Виміряйте пам'ять при обробці 100 000 слів: порівняйте пайплайн генераторів зі списковим підходом (tracemalloc або sys.getsizeof). Поясніть результат.

10.5. Базова асинхронна програма з asyncio.Реалізуйте асинхронну систему моніторингу «сервісів»:

- Функція check_service(name, url, interval) — корутина, що перевіряє сервіс кожні interval секунд (через asyncio.sleep) і видає статус 'OK' або 'ERROR' (випадковий збій 20% часу через random)
- Функція monitor(services) — запускає всі перевірки паралельно через asyncio.gather або create_task
- Логування з timestamp: кожне повідомлення починається з [HH:MM:SS]
- Автоматичне завершення через asyncio.wait_for з таймаутом 5 секунд
- Підрахунок загальної статистики: скільки разів кожен сервіс був OK/ERROR

Сервіси: ['auth-service', 'payment-api', 'database', 'cache', 'email-worker'], інтервали: [0.5, 0.7, 0.3, 0.4, 0.6].

## Контрольні запитання

1. Що таке протокол ітерації в Python? Які два методи повинен реалізовувати ітератор? Яка різниця між iterable і iterator?
2. Що відбувається всередині Python при виконанні циклу for x in some_list:? Опишіть кроки.
3. Що таке генератор? Чим генераторна функція відрізняється від звичайної? Що відбувається при виклику функції-генератора – вона одразу виконується?
4. Поясніть роботу ключового слова yield. Що відбувається з виконанням функції-генератора коли Python зустрічає yield?
5. Що таке yield from? Чим він відрізняється від циклу for з yield? Наведіть приклад рекурсивного сплощення списку.
6. Чим вираз-генератор відрізняється від спискового включення? Коли варто використовувати кожен варіант? Скільки пам'яті займає генератор на 1 мільйон елементів?
7. Назвіть три функції з модуля itertools і поясніть їхнє призначення. Чому itertools-функції є ефективнішими за власноруч написані цикли?
8. Що таке конкурентність (concurrency)? Чим вона відрізняється від паралелізму? Для яких задач підходить asyncio, а для яких – multiprocessing?
9. Поясніть ключові слова async def та await. Що таке корутина? Чи можна використовувати await поза async-функцією?
10. Яка різниця між asyncio.gather() та asyncio.create_task()? Коли await asyncio.sleep(1) краще, ніж time.sleep(1)?

Лабораторна робота №11.

## Робота з WEB-API за допомогою фреймворку FastAPI

**Мета роботи:** Зрозуміти принципи REST-архітектури та HTTP-протоколу; навчитися визначати маршрути FastAPI для обробки різних HTTP-методів (GET, POST, PUT, DELETE), а також описувати параметри шляху, параметри запиту та тіло запиту. Опанувати бібліотеку Pydantic для декларативної валідації даних: визначати схеми запитів і відповідей, використовувати анотації типів для автоматичної валідації та серіалізації/десеріалізації JSON. Навчитися захищати ендпоінти API за допомогою JWT-автентифікації, підключати SQLAlchemy для роботи з базою даних, обробляти помилки через HTTPException та тестувати API через автоматичну документацію Swagger UI.

### Теоретичні відомості:

### Що таке API і REST? Основні концепції

API (Application Programming Interface) – це набір правил, за якими одна програма може спілкуватися з іншою. Веб-API дозволяє клієнту (браузер, мобільний додаток, інша серверна програма) взаємодіяти з сервером через HTTP-запити.

REST (Representational State Transfer) – архітектурний стиль для веб-сервісів. API вважається RESTful якщо воно дотримується таких принципів:

- Клієнт-сервер – чіткий поділ між інтерфейсом клієнта і логікою сервера
- Без стану (stateless) – кожен запит містить усю необхідну інформацію, сервер не зберігає контекст між запитами
- Єдиний інтерфейс – URI однозначно ідентифікують ресурси; HTTP-методи описують дію
- Шаруватість – між клієнтом і сервером можуть бути проміжні ланки (кеш, балансувальник)

HTTP-методи та їхній смисл у REST:

| Метод | Призначення | Приклад URL | Дія |
|---|---|---|---|
| GET | Отримати дані | /students | Список студентів |
| GET | Отримати один | /students/{id} | Студент за ID |
| POST | Створити | /students | Новий студент |
| PUT | Замінити повністю | /students/{id} | Оновити студента |
| PATCH | Частково оновити | /students/{id} | Змінити поле |
| DELETE | Видалити | /students/{id} | Видалити студента |

HTTP-коди відповіді – сервер повідомляє клієнта про результат запиту числовим кодом:

| Код | Назва | Коли повертати |
|---|---|---|
| 200 OK | Успіх | Запит оброблено, дані у відповіді |
| 201 Created | Створено | POST: новий ресурс успішно створено |
| 204 No Content | Без вмісту | DELETE або PUT без тіла відповіді |
| 400 Bad Request | Погана відповідь | Помилка валідації, неправильний запит |
| 401 Unauthorized | Не авторизовано | Потрібна автентифікація |
| 403 Forbidden | Заборонено | Немає прав доступу |
| 404 Not Found | Не знайдено | Ресурс за вказаним ID не існує |
| 422 Unprocessable | Помилка даних | FastAPI: автоматично при помилці Pydantic |
| 500 Server Error | Помилка сервера | Необроблений виняток на сервері |

FastAPI – сучасний веб-фреймворк для Python, побудований на основі Starlette (ASGI) та Pydantic. Автоматично генерує інтерактивну документацію Swagger UI та ReDoc. Одна з найшвидших Python-бібліотек для веб-API – порівняна за продуктивністю з Go та Node.js.

```
# Встановлення
pip install fastapi uvicorn[standard] pydantic
pip install sqlalchemy python-jose[cryptography] passlib[bcrypt]

# Запуск сервера
uvicorn main:app --reload   # --reload: перезапуск при змінах коду
# Відкрийте: http://127.0.0.1:8000
# Swagger UI: http://127.0.0.1:8000/docs
# ReDoc:      http://127.0.0.1:8000/redoc
```

```
# main.py – мінімальний FastAPI-додаток
from fastapi import FastAPI

app = FastAPI(
    title='Студентський портал',
    description='API для управління студентами університету',
    version='1.0.0',
)

# GET /  – кореневий маршрут
@app.get('/')
def root():
    return {'message': 'Ласкаво просимо до API'}

# GET /health – перевірка стану сервісу
@app.get('/health')
def health_check():
    return {'status': 'ok', 'version': app.version}

# ── Структура відповіді – автоматично серіалізується у JSON ──
@app.get('/info')
def get_info():
    return {
        'name':  'MyApp',
        'items': [1, 2, 3],
        'active': True,
    }
```

## Параметри маршрутів. Path, Query та Body

Параметри шляху (Path parameters)

```
from fastapi import FastAPI, Path, HTTPException

app = FastAPI()

# ── Параметр шляху: {item_id} ──
@app.get('/students/{student_id}')
def get_student(student_id: int):   # тип int – автоматична валідація
    if student_id <= 0:
        raise HTTPException(status_code=404, detail='Студент не знайдений')
    return {'student_id': student_id, 'name': 'Іван'}

# Валідація через Path()
@app.get('/students/{student_id}/grades/{subject}')
def get_grade(
```

```
    student_id: int = Path(..., gt=0, le=9999, description='ID студента'),
    subject:    str = Path(..., min_length=2, max_length=50),
):
    return {'student_id': student_id, 'subject': subject, 'grade': 85}
```

Параметри запиту (Query parameters)

```
from typing import Optional
from fastapi import Query

@app.get('/students')
def list_students(
    skip:     int           = Query(0,    ge=0, description='Пропустити N записів'),
    limit:    int           = Query(10,   ge=1, le=100),
    dept:     Optional[str] = Query(None, description='Фільтр по кафедрі'),
    min_gpa:  float         = Query(0.0,  ge=0.0, le=5.0),
    sort_by:  str           = Query('name', regex='^(name|gpa|year)$'),
):
    # URL: GET /students?skip=0&limit=20&dept=CS&min_gpa=3.5
    students = fake_db[skip : skip + limit]
    if dept:
        students = [s for s in students if s['dept'] == dept]
    return {'total': len(students), 'items': students}

# Список значень: ?tag=python&tag=fastapi
@app.get('/courses')
def list_courses(
    tags: list[str] = Query(default=[], description='Фільтр по тегах')
):
    return {'tags': tags}
```

## Pydantic-моделі: валідація запитів і відповідей

Pydantic – бібліотека для декларативної валідації даних через анотації типів. FastAPI використовує Pydantic для автоматичної валідації тіла запиту, серіалізації відповіді та генерації JSON-схем.

```
from pydantic import BaseModel, Field, EmailStr, validator, field_validator
from typing import Optional
from datetime import datetime
from enum import Enum

class DeptEnum(str, Enum):
    CS       = 'CS'
    MATH     = 'Math'
    PHYSICS  = 'Physics'

# ── Схема для CREATE (вхідні дані) ──
class StudentCreate(BaseModel):
    name:   str   = Field(..., min_length=2, max_length=200,
                          description='Повне ім\'я студента')
    email:  EmailStr = Field(..., description='Email адреса')
    year:   int   = Field(..., ge=1, le=6, description='Рік навчання')
    gpa:    float = Field(0.0, ge=0.0, le=5.0)
    dept:   DeptEnum = DeptEnum.CS

    # Валідатор поля
    @field_validator('name')
    @classmethod
    def name_must_have_space(cls, v: str) -> str:
```

```
        if ' ' not in v.strip():
            raise ValueError('Вкажіть прізвище та ім\'я через пробіл')
        return v.strip()

# ── Схема для UPDATE (всі поля необов'язкові) ──
class StudentUpdate(BaseModel):
    name:  Optional[str]   = Field(None, min_length=2)
    year:  Optional[int]   = Field(None, ge=1, le=6)
    gpa:   Optional[float] = Field(None, ge=0, le=5)
    dept:  Optional[DeptEnum] = None

# ── Схема для ВІДПОВІДІ (включає id та created_at) ──
class StudentResponse(BaseModel):
    id:         int
    name:       str
    email:      str
    year:       int
    gpa:        float
    dept:       DeptEnum
    created_at: datetime

    model_config = {'from_attributes': True}  # читати з ORM-об'єктів

# ── Використання в маршрутах ──
from fastapi import FastAPI
from http import HTTPStatus

@app.post('/students', response_model=StudentResponse,
          status_code=201)
def create_student(student: StudentCreate):  # FastAPI сам парсить JSON body
    # Автоматична валідація: якщо дані невалідні – 422 Unprocessable Entity
    new_student = db_create(student.model_dump())
    return new_student

@app.put('/students/{sid}', response_model=StudentResponse)
def update_student(sid: int, data: StudentUpdate):
    student = db_get(sid)
    if not student:
        raise HTTPException(404, 'Не знайдено')
    updated = db_update(sid, data.model_dump(exclude_none=True))
    return updated
```

## CRUD API із SQLAlchemy

Повноцінний приклад: підключення FastAPI до SQLite через SQLAlchemy ORM.

```
# database.py
from sqlalchemy import create_engine
from sqlalchemy.orm import DeclarativeBase, sessionmaker, Session

SQLALCHEMY_URL = 'sqlite:///./students.db'
engine = create_engine(SQLALCHEMY_URL, connect_args={'check_same_thread': False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

class Base(DeclarativeBase):
    pass

# Dependency: генератор сесії
```

```python
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

```python
# models.py
from sqlalchemy import String, Integer, Float, DateTime, func
from sqlalchemy.orm import Mapped, mapped_column
from database import Base

class Student(Base):
    __tablename__ = 'students'
    id:         Mapped[int]    = mapped_column(primary_key=True, index=True)
    name:       Mapped[str]    = mapped_column(String(200))
    email:      Mapped[str]    = mapped_column(String(200), unique=True, index=True)
    year:       Mapped[int]    = mapped_column(Integer, default=1)
    gpa:        Mapped[float] = mapped_column(Float, default=0.0)
    dept:       Mapped[str]    = mapped_column(String(50))
    created_at: Mapped[DateTime] = mapped_column(DateTime, default=func.now())
# main.py – повний CRUD
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from sqlalchemy import select
from typing import List
from database import engine, get_db
from models import Student, Base
from schemas import StudentCreate, StudentUpdate, StudentResponse

Base.metadata.create_all(bind=engine)
app = FastAPI(title='Student API')

# ── GET /students – список ──
@app.get('/students', response_model=List[StudentResponse])
def list_students(
    skip: int = 0, limit: int = 10,
    db: Session = Depends(get_db)     # Depends: ін'єкція залежності
):
    students = db.execute(
        select(Student).offset(skip).limit(limit)
    ).scalars().all()
    return students

# ── GET /students/{id} – один ──
@app.get('/students/{student_id}', response_model=StudentResponse)
def get_student(student_id: int, db: Session = Depends(get_db)):
    student = db.get(Student, student_id)
    if not student:
        raise HTTPException(status_code=404, detail='Студент не знайдений')
    return student

# ── POST /students – створити ──
@app.post('/students', response_model=StudentResponse, status_code=201)
def create_student(data: StudentCreate, db: Session = Depends(get_db)):
    # Перевірка унікальності email
    existing = db.execute(
        select(Student).where(Student.email == data.email)
    ).scalar_one_or_none()
```

```
    if existing:
        raise HTTPException(status_code=400, detail='Email вже існує')
    student = Student(**data.model_dump())
    db.add(student)
    db.commit()
    db.refresh(student)
    return student

# ── PUT /students/{id} - оновити ──
@app.put('/students/{student_id}', response_model=StudentResponse)
def update_student(
    student_id: int,
    data: StudentUpdate,
    db: Session = Depends(get_db)
):
    student = db.get(Student, student_id)
    if not student:
        raise HTTPException(status_code=404, detail='Не знайдено')
    update_data = data.model_dump(exclude_none=True)
    for field, value in update_data.items():
        setattr(student, field, value)
    db.commit()
    db.refresh(student)
    return student

# ── DELETE /students/{id} ──
@app.delete('/students/{student_id}', status_code=204)
def delete_student(student_id: int, db: Session = Depends(get_db)):
    student = db.get(Student, student_id)
    if not student:
        raise HTTPException(status_code=404, detail='Не знайдено')
    db.delete(student)
    db.commit()
```

## Залежності (Depends) та Middleware

Система Depends дозволяє декларативно впроваджувати залежності у функції маршрутів – підключення до БД, перевірку автентифікації, логування тощо.

```
from fastapi import Depends, Header, Request
from typing import Annotated

# ── Залежність: пагінація ──
class PaginationParams:
    def __init__(self, skip: int = 0, limit: int = 10):
        self.skip  = max(0, skip)
        self.limit = min(100, limit)

PaginateDep = Annotated[PaginationParams, Depends(PaginationParams)]

@app.get('/courses')
def list_courses(pagination: PaginateDep, db: Session = Depends(get_db)):
    return db.execute(
        select(Course).offset(pagination.skip).limit(pagination.limit)
    ).scalars().all()

# ── Залежність: API-ключ у заголовку ──
API_KEY = 'secret-key-12345'

def verify_api_key(x_api_key: str = Header(...)):
```

```python
    if x_api_key != API_KEY:
        raise HTTPException(status_code=401, detail='Невірний API ключ')
    return x_api_key

# Захищений ендпоінт
@app.delete('/students/{sid}')
def delete_student(
    sid: int,
    db: Session = Depends(get_db),
    _: str = Depends(verify_api_key),   # перевірка ключа перед виконанням
):
    ...

# ── Middleware: логування часу відповіді ──
import time

@app.middleware('http')
async def log_requests(request: Request, call_next):
    start = time.perf_counter()
    response = await call_next(request)
    elapsed = time.perf_counter() - start
    print(f'{request.method} {request.url.path} – {elapsed:.3f}с')
    response.headers['X-Process-Time'] = str(elapsed)
    return response

# ── CORS (для фронтенд-запитів з іншого домену) ──
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=['http://localhost:3000', 'https://myapp.com'],
    allow_methods=['*'],
    allow_headers=['*'],
)
```

**JWT-автентифікація**

JWT (JSON Web Token) – стандарт для передачі підписаних токенів між клієнтом і сервером. Токен містить інформацію про користувача та підписаний секретним ключем. Сервер може перевірити токен без звернення до БД.

Структура JWT: Header.Payload.Signature – три частини Base64, з'єднані крапками.

```python
# auth.py – JWT автентифікація
from datetime import datetime, timedelta
from typing import Optional
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from jose import JWTError, jwt
from passlib.context import CryptContext
from pydantic import BaseModel

SECRET_KEY = 'your-secret-key-keep-in-env-variable'
ALGORITHM  = 'HS256'
ACCESS_TOKEN_EXPIRE_MINUTES = 30

pwd_context = CryptContext(schemes=['bcrypt'], deprecated='auto')
oauth2_scheme = OAuth2PasswordBearer(tokenUrl='/auth/token')

# ── Хешування паролів ──
```

```python
def hash_password(password: str) -> str:
    return pwd_context.hash(password)

def verify_password(plain: str, hashed: str) -> bool:
    return pwd_context.verify(plain, hashed)

# ── Генерація токена ──
class TokenData(BaseModel):
    access_token: str
    token_type:   str = 'bearer'

def create_access_token(data: dict,
                        expires_delta: Optional[timedelta] = None) -> str:
    to_encode = data.copy()
    expire = datetime.utcnow() + (expires_delta or timedelta(minutes=15))
    to_encode['exp'] = expire
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

# ── Залежність: отримати поточного користувача ──
def get_current_user(token: str = Depends(oauth2_scheme),
                     db: Session = Depends(get_db)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail='Не вдалося перевірити облікові дані',
        headers={'WWW-Authenticate': 'Bearer'},
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        email: str = payload.get('sub')
        if not email:
            raise credentials_exception
    except JWTError:
        raise credentials_exception
    user = get_user_by_email(db, email)
    if not user:
        raise credentials_exception
    return user

# ── Маршрути автентифікації ──
@app.post('/auth/register', status_code=201)
def register(data: UserCreate, db: Session = Depends(get_db)):
    if get_user_by_email(db, data.email):
        raise HTTPException(400, 'Email вже зареєстровано')
    hashed = hash_password(data.password)
    user = User(email=data.email, hashed_password=hashed, name=data.name)
    db.add(user); db.commit(); db.refresh(user)
    return {'id': user.id, 'email': user.email}

@app.post('/auth/token', response_model=TokenData)
def login(form: OAuth2PasswordRequestForm = Depends(),
          db: Session = Depends(get_db)):
    user = get_user_by_email(db, form.username)  # username = email
    if not user or not verify_password(form.password, user.hashed_password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail='Невірний email або пароль'
        )
    token = create_access_token(
        {'sub': user.email},
```

```
        expires_delta=timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    )
    return {'access_token': token, 'token_type': 'bearer'}

# ── Захищений маршрут ──
@app.get('/me', response_model=UserResponse)
def get_me(current_user: User = Depends(get_current_user)):
    return current_user

@app.get('/admin/stats')
def admin_stats(current_user: User = Depends(get_current_user),
                db: Session = Depends(get_db)):
    if current_user.role != 'admin':
        raise HTTPException(403, 'Тільки для адміністраторів')
    return {'total_users': db.query(User).count()}
```

**Роутери та структура проєкту**

Для великих API код розбивається на окремі модулі через APIRouter. Кожен модуль відповідає за свій ресурс.

```
# Структура проєкту
my_api/
├── main.py          # точка входу, підключення роутерів
├── database.py      # engine, SessionLocal, Base
├── models.py        # SQLAlchemy-моделі
├── schemas.py       # Pydantic-схеми
├── auth.py          # JWT автентифікація
├── config.py        # налаштування (секрети, URL бази)
└── routers/
    ├── students.py  # маршрути /students
    ├── courses.py   # маршрути /courses
    └── auth.py      # маршрути /auth
```

```
# routers/students.py
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from sqlalchemy import select
from typing import List
from database import get_db
from models import Student
from schemas import StudentCreate, StudentUpdate, StudentResponse
from auth import get_current_user

router = APIRouter(
    prefix='/students',      # всі маршрути: /students/...
    tags=['Students'],       # групування у Swagger UI
    responses={401: {'description': 'Не авторизовано'}},
)

@router.get('/', response_model=List[StudentResponse])
def list_students(db: Session = Depends(get_db)):
    return db.execute(select(Student)).scalars().all()

@router.get('/{student_id}', response_model=StudentResponse)
def get_student(student_id: int, db: Session = Depends(get_db)):
    s = db.get(Student, student_id)
    if not s: raise HTTPException(404, 'Не знайдено')
    return s
```

```python
@router.post('/', response_model=StudentResponse, status_code=201)
def create_student(
    data: StudentCreate,
    db:   Session = Depends(get_db),
    _:    object  = Depends(get_current_user),  # потрібна автентифікація
):
    s = Student(**data.model_dump())
    db.add(s); db.commit(); db.refresh(s)
    return s

# main.py
from fastapi import FastAPI
from routers import students, courses, auth as auth_router

app = FastAPI(title='University API')
app.include_router(students.router)
app.include_router(courses.router)
app.include_router(auth_router.router, prefix='/auth', tags=['Auth'])
```

Обробка помилок та валідація

```python
from fastapi import Request
from fastapi.responses import JSONResponse
from fastapi.exceptions import RequestValidationError
from pydantic import ValidationError

# ── Власний обробник помилок валідації (422) ──
@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request: Request, exc: RequestValidationError):
    errors = []
    for error in exc.errors():
        errors.append({
            'field':   ' → '.join(str(loc) for loc in error['loc']),
            'message': error['msg'],
            'type':    error['type'],
        })
    return JSONResponse(
        status_code=422,
        content={'detail': 'Помилка валідації', 'errors': errors}
    )

# ── HTTPException зі структурованою відповіддю ──
class APIError(Exception):
    def __init__(self, code: int, message: str, details: dict = None):
        self.code    = code
        self.message = message
        self.details = details or {}

@app.exception_handler(APIError)
async def api_error_handler(request: Request, exc: APIError):
    return JSONResponse(
        status_code=exc.code,
        content={'error': exc.message, 'details': exc.details}
    )

# ── Фонові задачі (Background Tasks) ──
from fastapi import BackgroundTasks
```

```python
def send_welcome_email(email: str, name: str):
    # Виконується після відповіді клієнту (не блокує відповідь!)
    print(f'Відправляємо email на {email}')

@app.post('/students', status_code=201)
def create_student(
    data: StudentCreate,
    background_tasks: BackgroundTasks,
    db: Session = Depends(get_db)
):
    student = Student(**data.model_dump())
    db.add(student); db.commit()
    # Email відправляється у фоні після відповіді
    background_tasks.add_task(send_welcome_email, data.email, data.name)
    return student

# ── Response з кастомним status code та заголовками ──
from fastapi import Response
from fastapi.responses import JSONResponse, RedirectResponse

@app.get('/students/{sid}/pdf')
def get_student_pdf(sid: int):
    pdf_bytes = generate_pdf(sid)  # умовна функція
    return Response(
        content=pdf_bytes,
        media_type='application/pdf',
        headers={'Content-Disposition': f'attachment; filename=student_{sid}.pdf'}
    )
```

Тестування FastAPI через TestClient

```python
# tests/test_students.py
import pytest
from fastapi.testclient import TestClient
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from main import app
from database import Base, get_db

# Тестова БД у пам'яті
TEST_DB_URL = 'sqlite://'
test_engine = create_engine(
    TEST_DB_URL, connect_args={'check_same_thread': False}
)
TestSession = sessionmaker(bind=test_engine)

@pytest.fixture(autouse=True)
def setup_db():
    Base.metadata.create_all(bind=test_engine)
    yield
    Base.metadata.drop_all(bind=test_engine)

@pytest.fixture
def client():
    def override_get_db():
        db = TestSession()
        try: yield db
        finally: db.close()
    app.dependency_overrides[get_db] = override_get_db
```

```python
    with TestClient(app) as c:
        yield c
    app.dependency_overrides.clear()

# ── Тести ──
def test_create_student(client):
    response = client.post('/students', json={
        'name': 'Іван Петренко',
        'email': 'ivan@test.com',
        'year': 2, 'gpa': 4.2, 'dept': 'CS'
    })
    assert response.status_code == 201
    data = response.json()
    assert data['name'] == 'Іван Петренко'
    assert 'id' in data

def test_get_student_not_found(client):
    response = client.get('/students/999')
    assert response.status_code == 404

def test_invalid_email(client):
    response = client.post('/students', json={
        'name': 'Test User', 'email': 'not-an-email',
        'year': 1, 'gpa': 3.0, 'dept': 'CS'
    })
    assert response.status_code == 422   # Pydantic validation error

def test_list_students_pagination(client):
    # Спочатку створюємо студентів
    for i in range(5):
        client.post('/students', json={
            'name': f'Student {i} Last', 'email': f's{i}@test.com',
            'year': 1, 'gpa': 3.0, 'dept': 'CS'
        })
    response = client.get('/students?skip=0&limit=3')
    assert response.status_code == 200
    assert len(response.json()) == 3
```

## Завдання на лабораторну роботу

Встановіть залежності:
`pip install fastapi uvicorn[standard] sqlalchemy pydantic[email] python-jose[cryptography] passlib[bcrypt] pytest httpx`
Для кожного завдання запускайте сервер через uvicorn та тестуйте API у Swagger UI (http://localhost:8000/docs).

11.1. Створіть файл main.py з FastAPI-додатком, що реалізує API для каталогу книг (зберігання у звичайному Python-словнику, без БД):

- `GET /books` – список всіх книг; параметри запиту: `skip, limit, genre` (фільтр), sort_by (title або year);
- `GET /books/{book_id}` – книга за ID; HTTPException 404 якщо не знайдено;
- `POST /books` – додати книгу (Pydantic-модель: title, author, year, genre, pages);
- `PUT /books/{book_id}` – замінити книгу повністю;

- `DELETE /books/{book_id}` – видалити книгу; 404 якщо немає;
- `GET /books/search` – пошук за параметром q (ilike по title або author).

Додайте валідацію: year між 1000 та поточним роком, pages > 0, title мінімум 2 символи. Відкрийте Swagger UI і протестуйте всі ендпоінти через браузер.

11.2. Pydantic-моделі та валідація. Розробіть схеми Pydantic для системи замовлень інтернет-магазину:

- `ProductCreate`: name (2-100 символів), price (> 0), category (Enum: electronics/clothing/food), stock (>= 0), description (Optional);
- ProductUpdate: всі поля Optional;
- ProductResponse: додає id, created_at, is_available (computed: stock > 0);
- OrderCreate: customer_email (EmailStr), items (List[OrderItem] де OrderItem: product_id та quantity > 0), delivery_address;
- OrderResponse: додає id, total_price (обчислюється з items), status (Enum: pending/confirmed/shipped/delivered).

Реалізуйте валідатори: total quantity в замовленні не більше 50 штук (@model_validator), discount якщо total > 1000 грн. Продемонструйте роботу через автоматичні тести pytest (мінімум 8 тестів).

11.3. Реалізуйте повноцінне CRUD API для системи управління завданнями (Task Manager) з підключенням до SQLite:

- Моделі: User (id, username, email), Project (id, name, owner_id FK), Task (id, title, description, status Enum, priority Enum, project_id FK, assignee_id FK, deadline datetime);
- CRUD для всіх трьох ресурсів: GET (список + пагінація), GET (один), POST, PUT, DELETE;
- Фільтри для задач: ?status=todo&priority=high&project_id=1;
- Логічне видалення: замість DELETE встановлюйте is_deleted=True;
- Обробка помилок: 404 для відсутніх ресурсів, 400 для дублювання email/username.

Напишіть мінімум 10 тестів через TestClient із тестовою SQLite в пам'яті.

11.4. JWT-автентифікація та захист ендпоінтів.

Додайте до завдання 3 повноцінну JWT-автентифікацію:

- POST /auth/register — реєстрація з хешуванням пароля через bcrypt
- POST /auth/token — отримати access token (форма OAuth2: username + password)
- GET /auth/me — інформація про поточного користувача (захищений)
- POST /auth/refresh — оновити токен (реалізуйте refresh token із терміном 7 днів)
- Ролі: user та admin; адміни можуть видаляти будь-які задачі, звичайні — лише свої
- Захищені ендпоінти: CREATE та DELETE вимагають автентифікації

Напишіть тести для auth flow: реєстрація → логін → отримання токена → захищений запит → помилка без токена.

11.5. Роутери та складна структура API. Розбийте API на модулі за допомогою APIRouter.

Реалізуйте університетський API з такою структурою:

```
university_api/
├── main.py
├── database.py
├── config.py        # BaseSettings з python-dotenv
├── models.py
```

```
├── schemas/
│   ├── student.py
│   ├── course.py
│   └── auth.py
└── routers/
    ├── students.py  # /students
    ├── courses.py   # /courses
    ├── grades.py    # /grades
    └── auth.py      # /auth
```

- students: CRUD + GET /students/{id}/courses (курси студента) + GET /students/{id}/transcript
- courses: CRUD + GET /courses/{id}/students + POST /courses/{id}/enroll
- grades: POST /grades (додати оцінку), GET /grades?student_id=&course_id=, PUT /grades/{id}
- Middleware: логування всіх запитів у файл access.log з timestamp, method, path, status, time_ms
- CORS налаштований для http://localhost:3000

11.6 Фонові задачі, Response та завантаження файлів. Розширте університетський API додатковими можливостями:

- POST /students/import — завантаження CSV-файлу з студентами через UploadFile; парсинг через Pandas; результат: {'imported': N, 'errors': [...]}
- GET /students/export — вивантаження всіх студентів у CSV; StreamingResponse із правильними заголовками
- POST /reports/generate — фонова задача: генерує PDF-звіт (імітація через time.sleep), відправляє email (print у консоль); відповідь одразу зі status 'queued'
- GET /stats — агрегована статистика: кількість студентів по кафедрах, середній GPA, топ-5 студентів (запит через SQLAlchemy aggregate functions)
- WebSocket /ws/notifications — простий WebSocket ендпоінт, що відправляє повідомлення підключеним клієнтам при додаванні нового студента

## Контрольні запитання

1. Що таке REST API? Назвіть чотири основних принципи REST-архітектури. Чим API відрізняється від звичайного веб-сайту?
2. Поясніть призначення HTTP-методів GET, POST, PUT, PATCH і DELETE у контексті REST. Яка різниця між PUT і PATCH?
3. Що таке Pydantic BaseModel? Яку роль відіграє Pydantic у FastAPI? Що відбувається якщо клієнт надсилає дані, що не проходять валідацію?
4. Поясніть різницю між Path parameter та Query parameter. Наведіть приклади URL для кожного випадку.
5. Що таке Depends() у FastAPI? Яку проблему вирішує система залежностей? Наведіть два приклади залежностей.
6. Яку роль відіграє response_model у декораторі @app.get()? Що відбувається з полями об'єкта, яких немає у response_model?
7. Що таке JWT? Яку структуру має JWT-токен? Поясніть різницю між access token та refresh token.
8. Що таке Middleware у FastAPI? Наведіть два практичних приклади використання middleware.
9. Як правильно тестувати FastAPI через TestClient? Чому важливо підміняти залежність get_db у тестах?
10. Навіщо розбивати API на роутери (APIRouter)? Яку перевагу дає сервісний шар (services/) порівняно з розміщенням логіки прямо у маршрутах?

Лабораторна робота №12.

**Робота з базами даних**

**Мета роботи:** Зрозуміти концепцію реляційних баз даних та мову SQL; навчитися підключатися до SQLite за допомогою стандартного модуля sqlite3, виконувати DDL та DML запити, використовувати параметризовані запити для захисту від SQL-ін'єкцій.
Опанувати бібліотеку SQLAlchemy: навчитися визначати ORM-моделі через декларативний підхід, виконувати CRUD-операції через сесії, будувати складні запити через Query API та описувати відносини між таблицями (один-до-багатьох, багато-до-багатьох).
Ознайомитися з концепцією міграцій бази даних (Alembic), навчитися безпечно вносити зміни у схему БД без втрати даних, а також застосовувати патерни репозиторію та Unit of Work для чистої архітектури коду.

**Теоретичні відомості:**

SQLite – компактна вбудована реляційна СУБД, що зберігає всю базу даних в одному файлі. Ідеальна для навчання, прототипування та невеликих застосунків. Модуль sqlite3 входить до стандартної бібліотеки Python.
Ключові поняття реляційних БД:

- Таблиця (table) – набір рядків і стовпців; кожен стовпець має тип і обмеження
- Первинний ключ (PRIMARY KEY) – унікальний ідентифікатор рядка
- Зовнішній ключ (FOREIGN KEY) – посилання на рядок іншої таблиці
- Транзакція – атомарна послідовність операцій: або всі виконуються, або жодна
- CRUD – Create, Read, Update, Delete – чотири базові операції з даними

```python
import sqlite3
from pathlib import Path

# ── Підключення до бази даних ──
# Файл БД (створюється автоматично якщо не існує)
conn = sqlite3.connect('university.db')

# БД у пам'яті (зникає після закриття – ідеально для тестів)
conn_memory = sqlite3.connect(':memory:')

# ── ОБОВ'ЯЗКОВО: закривати з'єднання ──
# Або використовувати менеджер контексту
with sqlite3.connect('university.db') as conn:
    cursor = conn.cursor()    # курсор для виконання запитів
    # ... запити ...
# З'єднання закривається автоматично

# ── Налаштування з'єднання ──
conn = sqlite3.connect('university.db')
conn.row_factory = sqlite3.Row   # рядки як dict-подібні об'єкти
conn.execute('PRAGMA foreign_keys = ON')  # вмикаємо зовнішні ключі!
```

***Важливо: PRAGMA foreign_keys***
*SQLite за замовчуванням НЕ перевіряє зовнішні ключі!*
*Завжди додавайте: conn.execute('PRAGMA foreign_keys = ON')*
*Або через cursor.execute() одразу після підключення.*

DDL (Data Definition Language) – команди для визначення структури БД: CREATE TABLE, ALTER TABLE, DROP TABLE

| Тип / Обмеження | Опис |
|---|---|
| INTEGER | Ціле число. AUTO INCREMENT для PRIMARY KEY. |
| REAL | Число з плаваючою точкою (8 байт). |
| TEXT | Рядок у кодуванні UTF-8. |
| BLOB | Бінарні дані (зображення, файли). |
| NULL | Відсутнє значення. |
| NOT NULL | Обмеження: значення обов'язкове. |
| UNIQUE | Обмеження: значення унікальне у стовпці. |
| DEFAULT val | Значення за замовчуванням. |
| CHECK(expr) | Перевірка умови при вставці/оновленні. |

```
import sqlite3

with sqlite3.connect(':memory:') as conn:
    conn.execute('PRAGMA foreign_keys = ON')

    # ── Створення таблиць ──
    conn.executescript('''
        CREATE TABLE IF NOT EXISTS departments (
            id       INTEGER PRIMARY KEY AUTOINCREMENT,
            name     TEXT    NOT NULL UNIQUE,
            building TEXT    NOT NULL
        );

        CREATE TABLE IF NOT EXISTS students (
            id          INTEGER PRIMARY KEY AUTOINCREMENT,
            name        TEXT    NOT NULL,
            email       TEXT    UNIQUE,
            year        INTEGER NOT NULL CHECK(year BETWEEN 1 AND 6),
            gpa         REAL    DEFAULT 0.0 CHECK(gpa BETWEEN 0 AND 5),
            dept_id     INTEGER REFERENCES departments(id) ON DELETE SET NULL,
            enrolled_at TEXT    DEFAULT (date('now'))
        );

        CREATE TABLE IF NOT EXISTS courses (
            id      INTEGER PRIMARY KEY AUTOINCREMENT,
            title   TEXT    NOT NULL,
            credits INTEGER NOT NULL CHECK(credits > 0),
            dept_id INTEGER REFERENCES departments(id)
        );

        -- Таблиця зв'язку: студент-курс (багато-до-багатьох)
        CREATE TABLE IF NOT EXISTS enrollments (
            student_id INTEGER REFERENCES students(id) ON DELETE CASCADE,
            course_id  INTEGER REFERENCES courses(id)  ON DELETE CASCADE,
            grade      REAL CHECK(grade BETWEEN 0 AND 100),
            enrolled   TEXT DEFAULT (date('now')),
            PRIMARY KEY (student_id, course_id)
        );
    ''')
    conn.commit()
    print('Схему БД створено успішно')

    # ── Перегляд схеми ──
    cur = conn.execute("SELECT name FROM sqlite_master WHERE type='table'")
    print('Таблиці:', [row[0] for row in cur.fetchall()])
```

DML (Data Manipulation Language) – команди для роботи з даними: INSERT, SELECT, UPDATE, DELETE.

Вставка та безпечні параметризовані запити

```python
import sqlite3

def get_db(path=':memory:'):
    conn = sqlite3.connect(path)
    conn.row_factory = sqlite3.Row
    conn.execute('PRAGMA foreign_keys = ON')
    return conn

conn = get_db('university.db')

# ── INSERT: одна строка ──
# НІКОЛИ не використовуйте f-рядки для підстановки даних!
# BAD:  f"INSERT INTO students VALUES ('{name}', ...)"  - SQL-ін'єкція!
# GOOD: параметризований запит із ?

conn.execute(
    'INSERT INTO departments (name, building) VALUES (?, ?)',
    ('Комп\'ютерні науки', 'ЦК')
)
conn.commit()

# ── INSERT: кілька рядків через executemany ──
students_data = [
    ('Іван Петренко',   'ivan@kpi.ua',  2, 4.2),
    ('Марія Коваль',    'maria@kpi.ua', 3, 4.8),
    ('Олег Шевченко',   'oleg@kpi.ua',  1, 3.5),
    ('Ліна Бондаренко', 'lina@kpi.ua',  2, 4.6),
    ('Тарас Мельник',   'taras@kpi.ua', 4, 3.9),
]
conn.executemany(
    'INSERT INTO students (name, email, year, gpa) VALUES (?, ?, ?, ?)',
    students_data
)
conn.commit()

# ── Отримати lastrowid ──
cur = conn.execute(
    'INSERT INTO courses (title, credits) VALUES (?, ?)',
    ('Програмування на Python', 4)
)
print(f'Новий курс ID: {cur.lastrowid}')
conn.commit()
```

SELECT: вибірка та фільтрація

```python
# ── Всі рядки ──
cur = conn.execute('SELECT * FROM students')
rows = cur.fetchall()           # список всіх рядків
print(f'Студентів: {len(rows)}')

# Завдяки row_factory = sqlite3.Row – доступ за іменем стовпця
for row in rows:
    print(f'{row["name"]:20} GPA: {row["gpa"]}')

# ── fetchone: один рядок ──
```

```
student = conn.execute(
    'SELECT * FROM students WHERE email = ?',
    ('ivan@kpi.ua',)
).fetchone()
print(student['name'] if student else 'Не знайдено')

# ── Фільтрація та сортування ──
good_students = conn.execute('''
    SELECT name, gpa
    FROM   students
    WHERE  gpa >= 4.0
    ORDER  BY gpa DESC
    LIMIT  3
''').fetchall()

# ── Агрегатні функції ──
stats = conn.execute('''
    SELECT
        COUNT(*)       AS total,
        AVG(gpa)       AS avg_gpa,
        MAX(gpa)       AS max_gpa,
        MIN(gpa)       AS min_gpa
    FROM students
''').fetchone()
print(f'Всього: {stats["total"]}, Середній GPA: {stats["avg_gpa"]:.2f}')

# ── JOIN: з'єднання таблиць ──
query = '''
    SELECT s.name, s.gpa, d.name AS dept
    FROM   students  s
    LEFT JOIN departments d ON s.dept_id = d.id
    WHERE  s.year = ?
    ORDER  BY s.gpa DESC
'''
second_year = conn.execute(query, (2,)).fetchall()

# ── GROUP BY ──
per_year = conn.execute('''
    SELECT year, COUNT(*) as cnt, AVG(gpa) as avg_gpa
    FROM   students
    GROUP  BY year
    HAVING COUNT(*) > 0
    ORDER  BY year
''').fetchall()
for row in per_year:
    print(f'Курс {row["year"]}: {row["cnt"]} студ., сер. GPA={row["avg_gpa"]:.2f}')
```

UPDATE та DELETE. Транзакції

```
# ── UPDATE ──
conn.execute(
    'UPDATE students SET gpa = ? WHERE email = ?',
    (4.5, 'ivan@kpi.ua')
)
affected = conn.execute(
    'UPDATE students SET year = year + 1 WHERE year < 6'
)
print(f'Оновлено рядків: {affected.rowcount}')
conn.commit()
```

```
# ── DELETE ──
conn.execute('DELETE FROM students WHERE gpa < 2.0')
conn.commit()

# ── Транзакції: все або нічого ──
def transfer_student(conn, student_id, new_dept_id):
    try:
        conn.execute('BEGIN')
        conn.execute(
            'UPDATE students SET dept_id = ? WHERE id = ?',
            (new_dept_id, student_id)
        )
        conn.execute(
            'INSERT INTO enrollments (student_id, course_id) VALUES (?, 1)',
            (student_id,)
        )
        conn.commit()          # підтверджуємо
        print('Переведення успішне')
    except sqlite3.Error as e:
        conn.rollback()        # скасовуємо всі зміни
        print(f'Помилка: {e}')

# ── Context manager для транзакцій ──
with conn:
    # всі операції всередині – одна транзакція
    # commit() при виході без помилок
    # rollback() при виключенні
    conn.execute('UPDATE students SET gpa = 4.0 WHERE id = 1')

# ── Закриття з'єднання ──
conn.close()
```

## Що таке ORM? Встановлення SQLAlchemy

ORM (Object-Relational Mapping) – технологія, що дозволяє працювати з базою даних через об'єкти Python, а не через рядки SQL. Таблиці стають класами, рядки – екземплярами, стовпці – атрибутами.

Переваги ORM:

- Код виглядає як Python, а не SQL – легше читати і підтримувати
- Незалежність від конкретної СУБД: один код працює з SQLite, PostgreSQL, MySQL
- Автоматичний захист від SQL-ін'єкцій
- Ліниве завантаження: пов'язані об'єкти завантажуються тільки при зверненні

```
# Встановлення
pip install sqlalchemy

# Для PostgreSQL: pip install sqlalchemy psycopg2-binary
# Для MySQL:      pip install sqlalchemy pymysql

import sqlalchemy
print(sqlalchemy.__version__)   # 2.x.x

# ── Рядки підключення (connection strings) ──
# SQLite (файл)
SQLITE_URL = 'sqlite:///university.db'
```

```
# SQLite (пам'ять – для тестів)
MEMORY_URL = 'sqlite://'

# PostgreSQL
PG_URL = 'postgresql://user:password@localhost:5432/dbname'

# MySQL
MYSQL_URL = 'mysql+pymysql://user:password@localhost/dbname'
```

## Декларативні моделі SQLAlchemy

Сучасний підхід SQLAlchemy 2.0 використовує DeclarativeBase. Кожна модель – клас Python, що успадковує від Base. Стовпці описуються через Mapped[тип] та mapped_column().

```
from sqlalchemy import (
    create_engine, String, Integer, Float, Text,
    ForeignKey, DateTime, Boolean, func
)
from sqlalchemy.orm import (
    DeclarativeBase, Mapped, mapped_column,
    relationship, Session
)
from datetime import datetime
from typing import Optional, List

# ── Базовий клас ──
class Base(DeclarativeBase):
    pass

# ── Модель Department ──
class Department(Base):
    __tablename__ = 'departments'

    id:       Mapped[int]           = mapped_column(primary_key=True)
    name:     Mapped[str]           = mapped_column(String(100), unique=True)
    building: Mapped[str]           = mapped_column(String(50))

    # Відношення один-до-багатьох: у кафедри багато студентів
    students: Mapped[List['Student']] = relationship(
        back_populates='department', cascade='all, delete-orphan'
    )
    courses: Mapped[List['Course']]   = relationship(back_populates='department')

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


# ── Модель Student ──
class Student(Base):
    __tablename__ = 'students'

    id:          Mapped[int]            = mapped_column(primary_key=True)
    name:        Mapped[str]            = mapped_column(String(200))
    email:       Mapped[Optional[str]]  = mapped_column(String(200), unique=True)
    year:        Mapped[int]            = mapped_column(Integer, default=1)
    gpa:         Mapped[float]          = mapped_column(Float, default=0.0)
    is_active:   Mapped[bool]           = mapped_column(Boolean, default=True)
    enrolled_at: Mapped[datetime]       = mapped_column(
        DateTime, default=func.now()
```

```
    )
    dept_id: Mapped[Optional[int]] = mapped_column(
        ForeignKey('departments.id', ondelete='SET NULL'), nullable=True
    )

    # Відношення
    department: Mapped[Optional[Department]] = relationship(
        back_populates='students'
    )
    enrollments: Mapped[List['Enrollment']] = relationship(
        back_populates='student', cascade='all, delete-orphan'
    )

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


# ── Таблиця зв'язку багато-до-багатьох ──
class Enrollment(Base):
    __tablename__ = 'enrollments'

    student_id: Mapped[int] = mapped_column(
        ForeignKey('students.id', ondelete='CASCADE'), primary_key=True
    )
    course_id: Mapped[int] = mapped_column(
        ForeignKey('courses.id', ondelete='CASCADE'), primary_key=True
    )
    grade:    Mapped[Optional[float]] = mapped_column(Float, nullable=True)
    enrolled: Mapped[datetime]        = mapped_column(DateTime, default=func.now())

    student: Mapped[Student] = relationship(back_populates='enrollments')
    course:  Mapped['Course'] = relationship(back_populates='enrollments')


class Course(Base):
    __tablename__ = 'courses'

    id:      Mapped[int]            = mapped_column(primary_key=True)
    title:   Mapped[str]            = mapped_column(String(200))
    credits: Mapped[int]            = mapped_column(Integer)
    dept_id: Mapped[Optional[int]]  = mapped_column(
        ForeignKey('departments.id'), nullable=True
    )

    department:  Mapped[Optional[Department]] = relationship(back_populates='courses')
    enrollments: Mapped[List[Enrollment]]     = relationship(back_populates='course')

    def __repr__(self) -> str:
        return f'Course(title={self.title!r}, credits={self.credits})'
```

Engine, Session та CRUD через ORM

```
from sqlalchemy import create_engine
from sqlalchemy.orm import Session

# ── Створення рушія ──
engine = create_engine(
    'sqlite:///university.db',
    echo=False,          # echo=True – виводить SQL у консоль (для дебагу)
```

```
)

# ── Створення таблиць ──
Base.metadata.create_all(engine)

# ── Сесія – одиниця роботи (Unit of Work) ──
# Всі зміни накопичуються у сесії і зберігаються через commit()
with Session(engine) as session:

    # ── CREATE ──
    dept = Department(name='Комп\'ютерні науки', building='ЦК')
    session.add(dept)
    session.flush()   # відправляє SQL, але не комітить – dept.id вже доступний

    students = [
        Student(name='Іван Петренко', email='ivan@kpi.ua',
                year=2, gpa=4.2, department=dept),
        Student(name='Марія Коваль', email='maria@kpi.ua',
                year=3, gpa=4.8, department=dept),
        Student(name='Олег Шевченко', email='oleg@kpi.ua',
                year=1, gpa=3.5, department=dept),
    ]
    session.add_all(students)
    session.commit()   # зберігаємо всі зміни
    print('Дані вставлено')

    # ── READ: session.get() – за первинним ключем ──
    student = session.get(Student, 1)   # або session.get(Student, id=1)
    print(student)

    # ── READ: SELECT через select() ──
    from sqlalchemy import select

    stmt = select(Student).where(Student.gpa >= 4.0).order_by(Student.gpa.desc())
    good_students = session.execute(stmt).scalars().all()
    for s in good_students:
        print(f'{s.name}: {s.gpa}')

    # ── UPDATE ──
    student = session.get(Student, 1)
    student.gpa = 4.5          # просто змінюємо атрибут
    student.year += 1
    session.commit()           # зміни зберігаються автоматично

    # ── DELETE ──
    low_performer = session.get(Student, 3)
    session.delete(low_performer)
    session.commit()

    print('CRUD операції виконано успішно')
```

Складні запити SQLAlchemy

```
from sqlalchemy import select, func, and_, or_, not_, desc, asc
from sqlalchemy.orm import Session, joinedload, selectinload

with Session(engine) as session:

    # ── Фільтрація: and_, or_, not_ ──
```

```python
    stmt = select(Student).where(
        and_(
            Student.gpa >= 4.0,
            Student.year.in_([2, 3]),
            Student.is_active == True
        )
    )
    results = session.execute(stmt).scalars().all()

    stmt2 = select(Student).where(
        or_(Student.gpa >= 4.5, Student.year == 1)
    )

    # ── LIKE та ilike (без урахування регістру) ──
    stmt3 = select(Student).where(Student.name.ilike('%іван%'))

    # ── Агрегація ──
    stmt4 = select(
        Student.year,
        func.count(Student.id).label('count'),
        func.avg(Student.gpa).label('avg_gpa'),
        func.max(Student.gpa).label('max_gpa')
    ).group_by(Student.year).order_by(Student.year)

    rows = session.execute(stmt4).all()
    for row in rows:
        print(f'Курс {row.year}: {row.count} студ., GPA={row.avg_gpa:.2f}')

    # ── JOIN ──
    stmt5 = (
        select(Student, Department.name.label('dept_name'))
        .join(Department, Student.dept_id == Department.id, isouter=True)
        .order_by(Student.gpa.desc())
    )
    results5 = session.execute(stmt5).all()

    # ── Eager loading: уникаємо N+1 запитів ──
    # Без eager loading: кожне звернення до student.department
    # генерує НОВИЙ SQL-запит!

    # joinedload: один JOIN-запит для всіх студентів із кафедрами
    stmt6 = select(Student).options(joinedload(Student.department))
    students = session.execute(stmt6).scalars().unique().all()
    for s in students:
        dept = s.department.name if s.department else 'Без кафедри'
        print(f'{s.name} – {dept}')  # 0 додаткових запитів!

    # selectinload: окремий запит IN (...) для колекцій
    stmt7 = select(Department).options(selectinload(Department.students))
    depts = session.execute(stmt7).scalars().all()
    for d in depts:
        print(f'{d.name}: {len(d.students)} студентів')

    # ── Subquery ──
    avg_gpa_sub = select(func.avg(Student.gpa)).scalar_subquery()
    stmt8 = select(Student).where(Student.gpa > avg_gpa_sub)
    above_avg = session.execute(stmt8).scalars().all()
    print(f'Вище середнього: {len(above_avg)}')
```

### Відносини між таблицями. Cascade

SQLAlchemy підтримує три типи відносин: один-до-одного, один-до-багатьох та багато-до-багатьох. Параметр cascade визначає, що відбувається з дочірніми об'єктами при видаленні батьківського.

```
# ── Один-до-багатьох: Department → Students ──
# Вже описано вище через relationship(back_populates=...)

with Session(engine) as session:
    # Доступ через відношення (не SQL!)
    dept = session.get(Department, 1)
    print(f'Студентів на кафедрі: {len(dept.students)}')
    for s in dept.students:
        print(f'  {s.name} ({s.gpa})')

    # Зворотне відношення
    student = session.get(Student, 1)
    print(f'Кафедра: {student.department.name}')

    # ── Багато-до-багатьох через Enrollment ──
    course = Course(title='Математичний аналіз', credits=5)
    session.add(course)
    session.flush()

    # Зарахувати студентів на курс
    enrollment1 = Enrollment(student_id=1, course_id=course.id, grade=85.0)
    enrollment2 = Enrollment(student_id=2, course_id=course.id, grade=92.0)
    session.add_all([enrollment1, enrollment2])
    session.commit()

    # Студенти курсу через JOIN
    from sqlalchemy import select
    stmt = (
        select(Student)
        .join(Enrollment, Enrollment.student_id == Student.id)
        .where(Enrollment.course_id == course.id)
    )
    enrolled = session.execute(stmt).scalars().all()
    print(f'Студентів на курсі: {len(enrolled)}')

    # ── cascade='all, delete-orphan' ──
    # При видаленні Department видаляються і всі його Students
    dept = session.get(Department, 1)
    session.delete(dept)
    session.commit()   # студенти видалені автоматично!
```

### Патерн Repository. Сесія як контекстний менеджер

Патерн Repository ізолює логіку доступу до БД від бізнес-логіки. Весь SQL і ORM-код зосереджений у репозиторії, а решта коду не знає, де зберігаються дані.

```
from sqlalchemy import create_engine, select, func
from sqlalchemy.orm import Session
from typing import Optional, List

class StudentRepository:
    '''Репозиторій для роботи зі студентами.'''
```

```python
    def __init__(self, session: Session):
        self._session = session

    def create(self, name: str, email: str, year: int, gpa: float) -> Student:
        student = Student(name=name, email=email, year=year, gpa=gpa)
        self._session.add(student)
        self._session.flush()   # отримуємо id без commit
        return student

    def get_by_id(self, student_id: int) -> Optional[Student]:
        return self._session.get(Student, student_id)

    def get_by_email(self, email: str) -> Optional[Student]:
        stmt = select(Student).where(Student.email == email)
        return self._session.execute(stmt).scalar_one_or_none()

    def get_all(self) -> List[Student]:
        stmt = select(Student).order_by(Student.name)
        return self._session.execute(stmt).scalars().all()

    def get_by_gpa_range(self, lo: float, hi: float) -> List[Student]:
        stmt = select(Student).where(
            Student.gpa.between(lo, hi)
        ).order_by(Student.gpa.desc())
        return self._session.execute(stmt).scalars().all()

    def update_gpa(self, student_id: int, new_gpa: float) -> bool:
        student = self.get_by_id(student_id)
        if not student:
            return False
        student.gpa = new_gpa
        return True

    def delete(self, student_id: int) -> bool:
        student = self.get_by_id(student_id)
        if not student:
            return False
        self._session.delete(student)
        return True

    def average_gpa(self) -> float:
        result = self._session.execute(
            select(func.avg(Student.gpa))
        ).scalar()
        return result or 0.0


# ── Використання через сесію ──
engine = create_engine('sqlite:///university.db')
Base.metadata.create_all(engine)

with Session(engine) as session:
    repo = StudentRepository(session)

    # Всі операції через репозиторій
    s = repo.create('Нова людина', 'new@kpi.ua', 1, 4.0)
    print(f'Створено: {s}')
```

```
    top = repo.get_by_gpa_range(4.5, 5.0)
    print(f'Відмінники: {[st.name for st in top]}')

    print(f'Сер. GPA: {repo.average_gpa():.2f}')

    session.commit()   # один commit для всього блоку
```

## Міграції з Alembic (огляд)

Alembic – інструмент міграцій для SQLAlchemy. Міграція – це файл з інструкціями, як змінити схему БД (додати стовпець, перейменувати таблицю тощо) без втрати даних.

```
# Встановлення
pip install alembic

# ── Ініціалізація ──
alembic init alembic
# Створює папку alembic/ та файл alembic.ini

# ── alembic.ini: вказати рядок підключення ──
# sqlalchemy.url = sqlite:///university.db

# ── alembic/env.py: вказати Base.metadata ──
# from models import Base
# target_metadata = Base.metadata

# ── Автогенерація міграції (порівнює моделі і БД) ──
alembic revision --autogenerate -m 'add phone column to students'

# ── Застосувати міграцію ──
alembic upgrade head

# ── Відкотити міграцію ──
alembic downgrade -1

# ── Переглянути поточну версію ──
alembic current

# ── Приклад файлу міграції (автогенерованого) ──
# versions/a1b2c3_add_phone_column.py

# def upgrade():
#     op.add_column('students',
#         sa.Column('phone', sa.String(20), nullable=True)
#     )
#
# def downgrade():
#     op.drop_column('students', 'phone')

# ── Ручна міграція ──
from alembic import op
import sqlalchemy as sa

def upgrade():
    # Додаємо стовпець
    op.add_column('students',
        sa.Column('phone', sa.String(20), nullable=True))
    # Додаємо індекс
    op.create_index('ix_students_year', 'students', ['year'])
```

```python
def downgrade():
    op.drop_index('ix_students_year', table_name='students')
    op.drop_column('students', 'phone')
```

## Завдання на лабораторну роботу

Для кожного завдання створіть окремий файл. Використовуйте SQLite як СУБД. Завдання 1–3 виконуйте через модуль sqlite3, завдання 4–7 – через SQLAlchemy.
Встановлення: `pip install sqlalchemy alembic`

12.1. Створіть БД library.db з таблицями authors (id, name, birth_year, country) та books (id, title, year, pages, genre, author_id FK). Виконайте:
- вставте 5 авторів та по 2–3 книги кожного через executemany;
- виведіть всі книги з іменем автора через JOIN;
- знайдіть автора з найбільшою кількістю книг (GROUP BY + ORDER BY + LIMIT 1);
- оновіть кількість сторінок однієї книги;
- видаліть всі книги жанру 'фантастика' і виведіть кількість видалених рядків;
- виведіть статистику по жанрах: назва жанру, кількість книг, середня кількість сторінок.

Використовуйте параметризовані запити для всіх DML-операцій.

12.2. Створіть БД tasks.db з таблицями projects (id, name, deadline TEXT, status) та tasks (id, title, description, priority TEXT, done INTEGER DEFAULT 0, project_id FK). Реалізуйте такі функції:
- create_project(conn, name, deadline) → int (повертає id);
- add_task(conn, project_id, title, priority) → int;
- complete_task(conn, task_id) – позначає завдання як виконане;
- get_project_stats(conn, project_id) → dict: total_tasks, done_tasks, percent_done;
- delete_project(conn, project_id) – видаляє проєкт і всі його задачі (транзакція);
- get_overdue(conn) → список проєктів де deadline < today і status != 'done'.

Продемонструйте коректну обробку помилок через try/except та rollback().

12.3. Створіть БД shop.db з таблицями customers, products та orders. Заповніть мінімум 20 записами. Реалізуйте запити:
- топ-5 клієнтів за сумою замовлень;
- найпопулярніший товар (за кількістю замовлень);
- місячна виручка за останні 6 місяців (GROUP BY strftime('%Y-%m', date));
- клієнти, що не робили замовлень за останній місяць (NOT IN або LEFT JOIN ... IS NULL);
- середній чек по категоріях товарів.

12.4. ORM-моделі та базовий CRUD (SQLAlchemy).
Реалізуйте систему управління гуртожитком. Визначте моделі:
- Building: id, name, address, floors
- Room: id, number, floor, capacity, building_id FK
- Resident: id, name, email, phone, room_id FK (nullable)
- Через SQLAlchemy Session виконайте:
- Створіть 2 будинки, по 5 кімнат у кожному, 15 мешканців

- Знайдіть всі вільні кімнати (де кількість мешканців < capacity) через subquery або join
- Переселіть мешканця в іншу кімнату (оновлення room_id)
- Знайдіть кімнату з найбільшою кількістю мешканців
- Видаліть будинок — всі кімнати і мешканці мають cascade видалення
- Виведіть список: будинок → кімнати → мешканці (eager loading через selectinload)

## Контрольні запитання

1. Що таке реляційна база даних? Поясніть поняття таблиці, рядка, стовпця, первинного і зовнішнього ключа.
2. Чому не можна використовувати f-рядки або конкатенацію рядків для підстановки змінних у SQL-запити? Що таке SQL-ін'єкція і як параметризовані запити захищають від неї?
3. Що таке транзакція? Поясніть властивості ACID. Для чого використовуються commit() і rollback()?
4. Чому після з'єднання з SQLite потрібно виконувати PRAGMA foreign_keys = ON? Що відбудеться без цієї команди при видаленні запису, на який є зовнішній ключ?
5. Що таке ORM? Назвіть три переваги ORM-підходу порівняно з прямими SQL-запитами.
6. Що таке сесія (Session) у SQLAlchemy? Чим session.flush() відрізняється від session.commit()?
7. Як в SQLAlchemy реалізується відношення один-до-багатьох? Що таке back_populates і для чого він потрібен?
8. Що таке проблема N+1 запитів? Як її вирішують joinedload та selectinload? В яких ситуаціях кожен підхід є кращим?
9. Що таке патерн Repository? Яку проблему він вирішує? Чому бізнес-логіка не повинна безпосередньо містити SQL або SQLAlchemy-запити?
10. Що таке міграція бази даних? Навіщо використовують Alembic? Поясніть різницю між командами alembic upgrade head та alembic downgrade -1.

Лабораторна робота №13.

# Веб-скрейпінг та автоматизація

**Мета роботи:** Зрозуміти принципи роботи HTTP-протоколу та структуру HTML-документів; навчитися виконувати GET і POST запити через бібліотеку requests, обробляти заголовки, куки та сесії; парсити HTML-сторінки за допомогою BeautifulSoup, використовуючи CSS-селектори та навігацію по дереву елементів. Ознайомитися з бібліотекою Selenium WebDriver для автоматизації браузера: відкриття сторінок, пошук елементів, взаємодія з формами та кнопками, очікування динамічного контенту; зрозуміти різницю між статичними та динамічними сайтами та вміти обирати правильний інструмент для кожного випадку.
Навчитися зберігати зібрані дані у структурованих форматах CSV та JSON, реалізовувати надійні павуки з обробкою помилок, дотриманням затримок між запитами та поважанням файлу robots.txt.

## Теоретичні відомості:

Веб-скрейпінг (web scraping) – автоматичне збирання даних із веб-сторінок. Це корисно коли потрібні дані розміщені на сайті, але сайт не надає публічного API.
Типові застосування:

- Аналіз цін конкурентів у інтернет-магазинах
- Збір новин та статей для аналізу тональності
- Моніторинг вакансій на сайтах пошуку роботи
- Збір наукових даних із відкритих баз знань
- Автоматизація рутинних дій у браузері (заповнення форм, тестування)

Вибір інструменту залежить від типу сайту:

- Статичний сайт (HTML у відповіді сервера) → requests + BeautifulSoup – швидко і просто
- Динамічний SPA (дані завантажуються через JavaScript/AJAX) → Selenium або Playwright
- Сайт з публічним API → requests + JSON-парсинг (найкращий варіант!)

***Етика веб-скрейпінгу – обов'язково дотримуйтесь:***
*1. Перевіряйте robots.txt: https://site.com/robots.txt*
*2. Дотримуйтесь затримок між запитами (1-3 секунди)*
*3. Не перевантажуйте сервер – максимум 1 запит/секунду*
*4. Читайте Terms of Service сайту – не всі дозволяють скрейпінг*
*5. Ідентифікуйте себе у User-Agent: 'MyBot/1.0 (contact@email.com)'*

## HTTP-запити через бібліотеку requests

Бібліотека `requests` – найпопулярніший інструмент для HTTP-запитів у Python. Вона приховує складність `urllib` та надає зручний, читабельний API.

# Встановлення

```
pip install requests beautifulsoup4 lxml selenium pandas
```

```python
import requests

# ── GET запит ──
response = requests.get('https://httpbin.org/get')

print(response.status_code)    # 200
print(response.headers)        # заголовки відповіді
print(response.text)           # тіло відповіді як рядок
print(response.json())         # розпарсений JSON
print(response.content)        # тіло як bytes (для файлів/зображень)
print(response.url)            # фінальний URL після перенаправлень
print(response.encoding)       # кодування (utf-8, cp1251...)

# Перевірка статусу — raise_for_status() кидає виняток при 4xx/5xx
response.raise_for_status()

# ── Параметри запиту (Query String) ──
params = {'q': 'python scraping', 'page': 1, 'limit': 20}
response = requests.get('https://api.example.com/search', params=params)
# URL стане: https://api.example.com/search?q=python+scraping&page=1&limit=20

# ── Заголовки та User-Agent ──
headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)',
    'Accept-Language': 'uk-UA,uk;q=0.9',
    'Accept': 'text/html,application/xhtml+xml',
}
response = requests.get('https://example.com', headers=headers)

# ── POST запит із JSON ──
data = {'username': 'user', 'password': 'pass'}
response = requests.post('https://api.example.com/login', json=data)

# ── POST із формою (application/x-www-form-urlencoded) ──
form_data = {'email': 'test@example.com', 'message': 'Hello'}
response = requests.post('https://example.com/contact', data=form_data)

# ── Таймаут ──
try:
    response = requests.get('https://slow-site.com', timeout=5)
except requests.exceptions.Timeout:
    print('Сервер не відповів за 5 секунд')
except requests.exceptions.ConnectionError:
    print('Помилка з\'єднання')
```

Сесії та куки

```python
import requests

# ── Session: зберігає куки і заголовки між запитами ──
session = requests.Session()

# Встановлюємо спільні заголовки для всіх запитів сесії
session.headers.update({
    'User-Agent': 'MyScraper/1.0',
    'Accept-Encoding': 'gzip, deflate',
})

# Логін (куки зберігаються автоматично)
login_resp = session.post('https://example.com/login',
```

```
    data={'user': 'admin', 'pass': '1234'})

# Тепер запити виконуються як авторизований користувач
profile = session.get('https://example.com/profile')
data    = session.get('https://example.com/api/data')

session.close()   # закрити сесію

# — Куки вручну —
cookies = {'session_id': 'abc123', 'theme': 'dark'}
response = requests.get('https://example.com', cookies=cookies)

# — Завантаження файлу (streaming) —
url = 'https://example.com/large-file.csv'
with requests.get(url, stream=True) as r:
    r.raise_for_status()
    with open('large-file.csv', 'wb') as f:
        for chunk in r.iter_content(chunk_size=8192):
            f.write(chunk)
print('Файл завантажено')
```

## Парсинг HTML через BeautifulSoup

BeautifulSoup перетворює HTML-рядок у дерево об'єктів, якими зручно навігувати та шукати елементи. Підтримує різні парсери: `html.parser` (вбудований), `lxml` (швидший), `html5lib` (точніший).

```
from bs4 import BeautifulSoup
import requests

# — Створення об'єкта —
html = requests.get('https://books.toscrape.com').text
soup = BeautifulSoup(html, 'html.parser')  # або 'lxml'

# — Базова навігація —
print(soup.title)          # <title>All products | ...</title>
print(soup.title.string)   # 'All products | ...'
print(soup.title.text)     # те саме

# — find(): перший збіг —
header = soup.find('h1')                    # перший <h1>
link   = soup.find('a', href=True)           # перший <a> з href
elem   = soup.find('div', class_='price')   # <div class='price'>
byid   = soup.find(id='main-content')        # за атрибутом id

# — find_all(): всі збіги → список —
all_links  = soup.find_all('a')
all_prices = soup.find_all('p', class_='price_color')
headings   = soup.find_all(['h1', 'h2', 'h3'])  # кілька тегів

print(f'Знайдено посилань: {len(all_links)}')
for p_tag in all_prices:
    print(p_tag.text.strip())

# — CSS-селектори через select() —
# select() повертає список; select_one() – перший або None
items    = soup.select('article.product_pod')       # article з класом
titles   = soup.select('h3 > a')                    # a безпосередньо у h3
```

```
ratings  = soup.select('[class^="star-rating"]')  # атрибут починається з
nav_item = soup.select_one('nav.breadcrumb li:last-child')

# ── Отримання атрибутів ──
for link in soup.find_all('a', limit=5):
    href = link.get('href')          # або link['href'] (KeyError якщо немає)
    text = link.get_text(strip=True) # текст без тегів
    print(f'{text}: {href}')

# ── Навігація по дереву ──
article = soup.find('article')
if article:
    print(article.parent.name)       # батьківський тег
    print(article.next_sibling)      # наступний сусід (може бути '\n'!)
    children = list(article.children)  # прямі нащадки
    print(article.find_next('p'))    # наступний <p> після article
```

**Практичний приклад: скрейпінг книжкового каталогу**

Сайт books.toscrape.com – спеціально призначений для практики скрейпінгу. Розглянемо повноцінний приклад збору даних із кількох сторінок.

```
import requests
from bs4 import BeautifulSoup
import csv
import json
import time

BASE_URL = 'https://books.toscrape.com/catalogue/'

RATING_MAP = {
    'One': 1, 'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5
}

def scrape_page(url: str) -> list[dict]:
    '''Збирає книги з однієї сторінки каталогу.'''
    headers = {'User-Agent': 'BookScraper/1.0'}
    try:
        response = requests.get(url, headers=headers, timeout=10)
        response.raise_for_status()
    except requests.RequestException as e:
        print(f'Помилка запиту: {e}')
        return []

    soup = BeautifulSoup(response.text, 'html.parser')
    books = []

    for article in soup.select('article.product_pod'):
        # Назва
        title_tag = article.select_one('h3 > a')
        title = title_tag['title'] if title_tag else 'N/A'

        # Ціна
        price_tag = article.select_one('p.price_color')
        price_str = price_tag.text.strip() if price_tag else '0'
        price = float(price_str.replace('Â', '').replace('£', '').strip())

        # Рейтинг
        rating_tag = article.select_one('p.star-rating')
        rating_word = rating_tag['class'][1] if rating_tag else 'Zero'
```

```
        rating = RATING_MAP.get(rating_word, 0)

        # Наявність
        avail_tag = article.select_one('p.availability')
        available = 'In stock' in avail_tag.text if avail_tag else False

        # Посилання на деталі
        link = BASE_URL + title_tag['href'].replace('../', '')

        books.append({
            'title':     title,
            'price':     price,
            'rating':    rating,
            'available': available,
            'url':       link,
        })

    return books


def scrape_all_pages(max_pages: int = 3) -> list[dict]:
    '''Обходить кілька сторінок каталогу.'''
    all_books = []

    for page in range(1, max_pages + 1):
        if page == 1:
            url = 'https://books.toscrape.com/catalogue/page-1.html'
        else:
            url = f'https://books.toscrape.com/catalogue/page-{page}.html'

        print(f'Скрейпінг сторінки {page}...')
        books = scrape_page(url)
        all_books.extend(books)

        time.sleep(1)   # ввічлива пауза між запитами

    return all_books


# ── Збереження результатів ──
books = scrape_all_pages(3)
print(f'Зібрано книг: {len(books)}')

# CSV
with open('books.csv', 'w', newline='', encoding='utf-8') as f:
    writer = csv.DictWriter(f, fieldnames=['title','price','rating','available','url'])
    writer.writeheader()
    writer.writerows(books)

# JSON
with open('books.json', 'w', encoding='utf-8') as f:
    json.dump(books, f, ensure_ascii=False, indent=2)

# Аналіз результатів
prices = [b['price'] for b in books]
print(f'Середня ціна:  £{sum(prices)/len(prices):.2f}')
print(f'Макс ціна:     £{max(prices):.2f}')
print(f'5-зіркових:    {sum(1 for b in books if b["rating"]==5)}')
```

## Робота з API через requests + JSON

Якщо сайт надає публічний API – завжди краще використовувати його замість парсингу HTML. API повертає чисті дані у форматі JSON без зайвої розмітки.

```python
import requests
import json
from pprint import pprint

# ── Публічний REST API (JSONPlaceholder - тестовий API) ──
BASE = 'https://jsonplaceholder.typicode.com'

# GET список
posts = requests.get(f'{BASE}/posts').json()
print(f'Постів: {len(posts)}')
pprint(posts[0])

# GET один елемент
post = requests.get(f'{BASE}/posts/1').json()
print(post['title'])

# GET з параметрами фільтрації
user_posts = requests.get(f'{BASE}/posts', params={'userId': 1}).json()
print(f'Постів користувача 1: {len(user_posts)}')

# ── Обробка пагінованих API ──
def fetch_all_pages(base_url: str, per_page: int = 30) -> list:
    '''Збирає всі сторінки пагінованого API.'''
    all_items = []
    page = 1

    while True:
        response = requests.get(base_url, params={
            'page': page,
            'per_page': per_page,
        })
        response.raise_for_status()
        items = response.json()

        if not items:          # порожня відповідь = кінець даних
            break

        all_items.extend(items)
        print(f'Сторінка {page}: отримано {len(items)} записів')

        # Деякі API вказують загальну кількість у заголовку
        total = int(response.headers.get('X-Total-Count', 0))
        if total and len(all_items) >= total:
            break

        page += 1
        time.sleep(0.5)

    return all_items

# ── API з автентифікацією ──
GITHUB_TOKEN = 'your_token_here'

headers = {
    'Authorization': f'Bearer {GITHUB_TOKEN}',
    'Accept': 'application/vnd.github+json',
```

```
}

repos = requests.get(
    'https://api.github.com/users/octocat/repos',
    headers=headers
).json()

for repo in repos[:5]:
    print(f'{repo["name"]}: {repo["stargazers_count"]} stars')
```

## Selenium WebDriver: автоматизація браузера

Selenium керує реальним браузером через WebDriver. Ідеальний для динамічних сайтів, де JavaScript генерує контент після завантаження сторінки.

# Встановлення

```
pip install selenium
```

*# Selenium 4.6+ автоматично завантажує потрібний ChromeDriver!*

*# Не потрібно вручну завантажувати chromedriver.exe*

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.keys import Keys
import time

# ── Налаштування Chrome ──
options = Options()
options.add_argument('--headless')          # без відкриття вікна браузера
options.add_argument('--no-sandbox')
options.add_argument('--disable-dev-shm-usage')
options.add_argument('--window-size=1920,1080')
options.add_argument('user-agent=Mozilla/5.0 ...')

# ── Запуск браузера ──
driver = webdriver.Chrome(options=options)

try:
    # ── Відкрити сторінку ──
    driver.get('https://www.wikipedia.org')
    print(driver.title)     # 'Wikipedia'

    # ── Пошук елементів ──
    # By.ID, By.CLASS_NAME, By.TAG_NAME
    # By.CSS_SELECTOR, By.XPATH, By.NAME, By.LINK_TEXT

    # Знайти поле пошуку і ввести текст
    search_box = driver.find_element(By.NAME, 'search')
    search_box.clear()
    search_box.send_keys('Python programming language')
    search_box.send_keys(Keys.RETURN)    # натиснути Enter
```

```
    # ── Явне очікування (краще ніж time.sleep!) ──
    wait = WebDriverWait(driver, timeout=10)
    heading = wait.until(
        EC.presence_of_element_located((By.TAG_NAME, 'h1'))
    )
    print(heading.text)   # 'Python (programming language)'

    # ── Отримати HTML і передати у BeautifulSoup ──
    from bs4 import BeautifulSoup
    soup = BeautifulSoup(driver.page_source, 'lxml')
    first_para = soup.select_one('#mw-content-text p')
    print(first_para.get_text()[:200])

finally:
    driver.quit()   # ОБОВ'ЯЗКОВО закривати браузер!
```

Selenium: взаємодія з елементами та JavaScript

| Метод / Синтаксис | Опис / Результат |
|---|---|
| `find_element(By.ID, 'id')` | Знайти елемент за id |
| `find_element(By.CLASS_NAME, 'c')` | Знайти за класом CSS |
| `find_element(By.CSS_SELECTOR, s)` | Знайти за CSS-селектором |
| `find_element(By.XPATH, path)` | Знайти за XPath-виразом |
| `find_elements(...)` | Список всіх збігів (не один!) |
| `elem.click()` | Клікнути по елементу |
| `elem.send_keys(text)` | Ввести текст у поле |
| `elem.clear()` | Очистити поле введення |
| `elem.text` | Текст всередині елемента |
| `elem.get_attribute('href')` | Значення атрибута |
| `elem.is_displayed()` | Чи видимий елемент |
| `elem.is_enabled()` | Чи активний (не disabled) |
| `driver.execute_script(js)` | Виконати JavaScript-код |
| `driver.save_screenshot(path)` | Зберегти скрішнот сторінки |

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import Select, WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.action_chains import ActionChains

driver = webdriver.Chrome()
wait   = WebDriverWait(driver, 10)

driver.get('https://example-form.com')

# ── Заповнення форми ──
driver.find_element(By.ID, 'name').send_keys('Іван Петренко')
driver.find_element(By.ID, 'email').send_keys('ivan@test.com')

# ── Вибір зі списку (select) ──
select_elem = driver.find_element(By.ID, 'city')
select = Select(select_elem)
select.select_by_visible_text('Київ')    # за текстом
select.select_by_value('kyiv')            # за значенням
select.select_by_index(2)                 # за позицією
```

```python
# ── Прапорці та радіо-кнопки ──
checkbox = driver.find_element(By.ID, 'agree')
if not checkbox.is_selected():
    checkbox.click()

# ── Scroll ──
driver.execute_script('window.scrollTo(0, document.body.scrollHeight)')
driver.execute_script('arguments[0].scrollIntoView(true);', checkbox)

# ── JavaScript: клік на прихований елемент ──
driver.execute_script('arguments[0].click();', hidden_elem)

# ── Hover (навести мишу) ──
actions = ActionChains(driver)
actions.move_to_element(menu_item).perform()

# ── Очікування зникнення елемента (loader) ──
wait.until(EC.invisibility_of_element_located(
    (By.CLASS_NAME, 'loading-spinner')
))

# ── Очікування кліку ──
button = wait.until(
    EC.element_to_be_clickable((By.CSS_SELECTOR, 'button[type=submit]'))
)
button.click()

# ── Скріншот ──
driver.save_screenshot('result.png')

driver.quit()
```

**Playwright – сучасна альтернатива Selenium**

Playwright від Microsoft – новіша бібліотека для автоматизації браузера. Підтримує Chromium, Firefox та WebKit. Має вбудовані інтелектуальні очікування та більш зручний API.

# Встановлення
```
pip install playwright
playwright install chromium      # завантажити браузер
```

```python
from playwright.sync_api import sync_playwright

with sync_playwright() as p:
    # ── Запуск браузера ──
    browser = p.chromium.launch(headless=True)
    page    = browser.new_page()

    # ── Навігація ──
    page.goto('https://books.toscrape.com')
    print(page.title())

    # ── Пошук і взаємодія ──
    page.fill('#search', 'python')   # знайти input і ввести текст
```

```
    page.click('button[type=submit]')

    # ── Інтелектуальне очікування (автоматично!) ──
    # Playwright сам чекає поки елемент стане видимим
    titles = page.locator('h3 > a').all_text_contents()
    print(titles[:5])

    # ── CSS-селектори та XPath ──
    prices = page.locator('p.price_color').all_text_contents()

    # ── Скріншот ──
    page.screenshot(path='screenshot.png', full_page=True)

    # ── Отримати HTML для BeautifulSoup ──
    from bs4 import BeautifulSoup
    soup = BeautifulSoup(page.content(), 'lxml')

    browser.close()

# ── Асинхронна версія ──
import asyncio
from playwright.async_api import async_playwright

async def scrape_async():
    async with async_playwright() as p:
        browser = await p.chromium.launch(headless=True)
        page    = await browser.new_page()
        await page.goto('https://example.com')
        content = await page.content()
        await browser.close()
        return content

asyncio.run(scrape_async())
```

Надійний павук: обробка помилок, затримки, robots.txt

```
import requests
import time
import random
from bs4 import BeautifulSoup
from urllib.robotparser import RobotFileParser
from urllib.parse import urljoin, urlparse
from dataclasses import dataclass, field, asdict
import csv
import logging

# ── Налаштування логування ──
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s',
    handlers=[
        logging.FileHandler('scraper.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)


@dataclass
```

```python
class ScraperConfig:
    base_url:      str
    delay_min:     float = 1.0     # мінімальна затримка між запитами
    delay_max:     float = 3.0     # максимальна затримка
    max_retries:   int   = 3       # кількість повторних спроб
    timeout:       int   = 10
    user_agent:    str   = 'MyScraper/1.0 (educational project)'


class RobustScraper:
    def __init__(self, config: ScraperConfig):
        self.config  = config
        self.session = requests.Session()
        self.session.headers['User-Agent'] = config.user_agent
        self._robot_parser = None

    def _check_robots(self, url: str) -> bool:
        '''Перевіряє чи дозволений скрейпінг для цього URL.'''
        if not self._robot_parser:
            parsed    = urlparse(self.config.base_url)
            robots_url = f'{parsed.scheme}://{parsed.netloc}/robots.txt'
            self._robot_parser = RobotFileParser()
            self._robot_parser.set_url(robots_url)
            try:
                self._robot_parser.read()
            except Exception:
                return True  # якщо robots.txt недоступний – дозволяємо
        return self._robot_parser.can_fetch(self.config.user_agent, url)

    def fetch(self, url: str) -> str | None:
        '''Завантажує сторінку з повторними спробами.'''
        if not self._check_robots(url):
            logger.warning(f'robots.txt забороняє: {url}')
            return None

        for attempt in range(1, self.config.max_retries + 1):
            try:
                logger.info(f'Запит {attempt}/{self.config.max_retries}: {url}')
                response = self.session.get(url, timeout=self.config.timeout)
                response.raise_for_status()

                # Ввічлива затримка
                delay = random.uniform(self.config.delay_min, self.config.delay_max)
                time.sleep(delay)

                return response.text

            except requests.exceptions.HTTPError as e:
                if e.response.status_code == 429:  # Too Many Requests
                    wait = 2 ** attempt  # exponential backoff
                    logger.warning(f'Rate limit. Чекаємо {wait}с...')
                    time.sleep(wait)
                elif e.response.status_code == 404:
                    logger.error(f'Сторінка не знайдена: {url}')
                    return None
                else:
                    logger.error(f'HTTP помилка: {e}')

            except requests.exceptions.RequestException as e:
```

```
                logger.error(f'Помилка запиту (спроба {attempt}): {e}')
                if attempt < self.config.max_retries:
                    time.sleep(2 ** attempt)

        logger.error(f'Всі спроби вичерпано: {url}')
        return None

    def close(self):
        self.session.close()
```

Збереження даних: CSV, JSON, Pandas

```
import csv
import json
import pandas as pd
from pathlib import Path
from datetime import datetime

# ── Збереження у CSV ──
books = [
    {'title': 'Python Crash Course', 'price': 29.99, 'rating': 5},
    {'title': 'Clean Code',          'price': 35.00, 'rating': 4},
]

output_dir = Path('scraped_data')
output_dir.mkdir(exist_ok=True)

csv_path = output_dir / 'books.csv'
with open(csv_path, 'w', newline='', encoding='utf-8') as f:
    if books:
        writer = csv.DictWriter(f, fieldnames=books[0].keys())
        writer.writeheader()
        writer.writerows(books)
print(f'Збережено {len(books)} записів у {csv_path}')

# ── Збереження у JSON із метаданими ──
result = {
    'metadata': {
        'scraped_at': datetime.now().isoformat(),
        'source':     'books.toscrape.com',
        'total':      len(books),
    },
    'data': books,
}
json_path = output_dir / 'books.json'
with open(json_path, 'w', encoding='utf-8') as f:
    json.dump(result, f, ensure_ascii=False, indent=2)

# ── Обробка через Pandas ──
df = pd.DataFrame(books)
print(df.describe())
print(df.sort_values('price', ascending=False).head(10))

# Дедублікація
df = df.drop_duplicates(subset=['title'])

# Збереження у Excel
df.to_excel(output_dir / 'books.xlsx', index=False, sheet_name='Books')
```

```python
# ── Дописування у існуючий CSV ──
def append_to_csv(filepath: Path, new_rows: list[dict]):
    '''Дописує рядки у CSV без перезапису.'''
    if not new_rows:
        return
    file_exists = filepath.exists()
    with open(filepath, 'a', newline='', encoding='utf-8') as f:
        writer = csv.DictWriter(f, fieldnames=new_rows[0].keys())
        if not file_exists:
            writer.writeheader()  # заголовок тільки для нового файлу
        writer.writerows(new_rows)
```

**Завдання на дипломну роботу**

Встановіть залежності:

```
pip install requests beautifulsoup4 lxml selenium playwright pandas openpyxl
```

Для практики скрейпінгу використовуйте дозволені тренувальні сайти: books.toscrape.com, quotes.toscrape.com, httpbin.org, jsonplaceholder.typicode.com

13.1. HTTP-запити та базовий парсинг. Виконайте серію завдань з бібліотекою requests:

- GET запит до https://httpbin.org/get із власними заголовками (User-Agent, Accept-Language); виведіть статус-код, заголовки відповіді та тіло.
- GET запит до https://jsonplaceholder.typicode.com/users – отримати список всіх користувачів; вивести ім'я та email кожного.
- Завантажити зображення з https://httpbin.org/image/png і зберегти у файл image.png (stream=True).
- POST запит до https://httpbin.org/post із JSON-тілом: ваше ім'я, дата, довільні дані; вивести що сервер отримав.
- Реалізувати функцію safe_get(url, retries=3) з обробкою ConnectionError, Timeout та HTTPError.

13.2. BeautifulSoup: парсинг HTML. Зберіть дані з сайту https://quotes.toscrape.com:

- Отримайте першу сторінку; знайдіть і виведіть всі цитати (.text), авторів (.author) та теги (.tags).
- Знайдіть посилання на сторінки авторів (теги <a> в .author); перейдіть на сторінку першого автора і зберіть біографію (ім'я, дата народження, місце народження).
- Обійдіть перші 3 сторінки (кожна: /page/2/, /page/3/); зберіть всі цитати у список словників: text, author, tags.
- Збережіть результат у quotes.csv та quotes.json.
- Підрахуйте топ-5 авторів за кількістю цитат та топ-5 найуживаніших тегів.

13.3. Парсинг книжкового каталогу. Використайте сайт https://books.toscrape.com. Реалізуйте клас BookScraper:

- Метод scrape_page(url) → список книг (title, price, rating, availability, category, url_detail).
- Метод scrape_category(category_url) → всі книги категорії (з обходом всіх сторінок пагінації).
- Метод get_book_details(url) → деталі книги (UPC, тип товару, ціна без ПДВ, ціна з ПДВ, кількість на складі, опис).
- Затримка 1–2 секунди між запитами; обробка помилок; логування у файл.

- Збережіть мінімум 50 книг у books.csv та books.xlsx (через Pandas).

13.4. Selenium: автоматизація динамічного сайту. Використайте Selenium для роботи з https://quotes.toscrape.com/scroll (версія з нескінченним скролом):

- Відкрийте сторінку у headless-режимі.
- Прокручуйте сторінку донизу через execute_script поки не завантажаться нові цитати (очікуйте появи нових елементів через WebDriverWait).
- Зберіть мінімум 50 цитат (прокручуйте стільки разів скільки потрібно).
- Зробіть скрішнот сторінки після прокрутки.
- Передайте page.page_source у BeautifulSoup для фінального парсингу.
- Порівняйте результат із requests+BS4 підходом: що відрізняється?

13.5. Автоматизація форм та взаємодія з елементами. Використайте сайт https://quotes.toscrape.com/login для автоматизації входу:

- Відкрийте сторінку входу через Selenium
- Знайдіть поля username та password, введіть дані (user: admin, pass: admin або будь-яка комбінація)
- Натисніть кнопку Login; перевірте чи вхід успішний (наявність певного елемента на сторінці)
- Реалізуйте автоматичне заповнення форми додавання цитати (якщо є)
- Зберіть список цитат після входу (можуть відрізнятись від публічного списку)

Додатково: реалізуйте автоматизацію через https://the-internet.herokuapp.com/ – сторінки: form authentication, dropdowns, checkboxes, file upload.

## Контрольні запитання

1. Що таке веб-скрейпінг? Назвіть три реальних застосування. Які етичні правила необхідно дотримуватись?
2. Для чого призначений файл robots.txt? Як перевірити його вміст і чому важливо його враховувати?
3. Чим відрізняються методи requests.get() та requests.Session().get()? Коли Session є кращим вибором?
4. Що таке User-Agent і навіщо його встановлювати при скрейпінгу? Що може статись якщо надсилати запити без User-Agent?
5. Поясніть різницю між find() та find_all() у BeautifulSoup. Яку перевагу дають CSS-селектори через select() порівняно з find()?
6. Для чого потрібен Selenium якщо є requests + BeautifulSoup? На яких типах сайтів requests не спрацює?
7. Що таке WebDriverWait та EC.presence_of_element_located()? Чому явне очікування краще ніж time.sleep()?
8. Що таке exponential backoff? Поясніть алгоритм на прикладі: як він допомагає при отриманні помилки 429 (Too Many Requests)?
9. Назвіть три способи знайти елемент у Selenium (By.ID, By.CSS_SELECTOR тощо). Який спосіб є найнадійнішим і чому?
10. Яка різниця між Selenium та Playwright? Назвіть дві переваги Playwright над Selenium для сучасних веб-додатків.

Лабораторна робота №14.

**Бібліотеки NumPy та Matplotlib**

**Мета роботи:** Ознайомитися з бібліотекою NumPy. Навчитися створювати масиви, виконувати векторизовані математичні операції, розв'язувати задачі лінійної алгебри та статистики, які стануть у нагоді при вивченні фізики, математики та інженерних дисциплін. Навчитися будувати та форматувати графіки за допомогою бібліотеки Matplotlib. Сформувати практичні навички збереження графіків у файли різних форматів, а також застосовувати набуті знання для вирішення прикладних задач із фізики, математики та аналізу даних.

## Теоретичні відомості:

### Частина І. Бібліотека NumPy.

`NumPy` (Numerical Python) – фундаментальна бібліотека наукових обчислень у Python. Її основа – об'єкт `ndarray` (N-dimensional array): багатовимірний масив однорідних (одного типу) елементів.

*Чому NumPy швидший за звичайні списки Python?*
*Елементи масиву зберігаються у неперервній ділянці пам'яті (на відміну від списків). Операції виконуються у скомпільованому C/Fortran-коді без інтерпретатора Python.*

Векторизація: одна операція застосовується одночасно до всіх елементів

```python
import numpy as np

# ── Створення масивів ──

# Зі списку Python
a = np.array([1, 2, 3, 4, 5])
print(a)          # [1 2 3 4 5]
print(type(a))    # <class 'numpy.ndarray'>

# Двовимірний масив (матриця) зі вкладеного списку
m = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

# Ключові атрибути масиву
print(a.shape)     # (5,)     – форма (кортеж розмірностей)
print(m.shape)     # (3, 3)   – 3 рядки, 3 стовпці
print(m.ndim)      # 2        – кількість вимірів
print(m.size)      # 9        – загальна кількість елементів
print(m.dtype)     # int64    – тип даних
print(m.itemsize)  # 8        – байт на елемент
```

Функції створення масивів

| Синтаксис | Опис / Результат |
|---|---|
| `np.zeros((m, n))` | Масив нулів форми m×n: `np.zeros((2,3))` |
| `np.ones((m, n))` | Масив одиниць: `np.ones((3,3))` |
| `np.full((m,n), val)` | Масив, заповнений `val`: `np.full((2,4), 7)` |
| `np.eye(n)` | Одинична матриця `n×n` (діагональ = 1) |

| | |
|---|---|
| `np.arange(s,e,step)` | Масив чисел у діапазоні з кроком (як `range()`) |
| `np.linspace(s,e,n)` | `n` рівномірно розміщених точок від `s` до `e` (включно) |
| `np.logspace(s,e,n)` | `n` точок у логарифмічній шкалі від `10^s` до `10^e` |
| `np.random.rand(m,n)` | Випадкові числа `[0,1)` рівномірного розподілу |
| `np.random.randn(m,n)` | Випадкові числа нормального розподілу `N(0,1)` |
| `np.random.randint(l,h,n)` | `n` випадкових цілих чисел від `l` до `h-1` |

```python
# arange – аналог range(), але повертає масив
x = np.arange(0, 10, 2)       # [0 2 4 6 8]
x = np.arange(0.0, 1.1, 0.25) # [0.   0.25 0.5  0.75 1.  ]

# linspace – рівномірна сітка точок (ідеальна для графіків!)
x = np.linspace(0, 2*np.pi, 100)  # 100 точок від 0 до 2π

# ones, zeros, eye
print(np.zeros((2, 4)))
# [[0. 0. 0. 0.]
#  [0. 0. 0. 0.]]

print(np.eye(3))
# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]

# Вказання типу даних
a = np.array([1, 2, 3], dtype=np.float64)
b = np.zeros(5, dtype=np.int32)
print(a.dtype)   # float64
```

Індексування, зрізи та форма масивів

```python
import numpy as np

# ── Одновимірний масив ──
a = np.array([10, 20, 30, 40, 50])
print(a[0])      # 10  – перший елемент
print(a[-1])     # 50  – останній
print(a[1:4])    # [20 30 40]  – зріз
print(a[::2])    # [10 30 50]  – кожен другий

# ── Двовимірний масив (матриця) ──
m = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

print(m[0, 0])   # 1  – рядок 0, стовпець 0
print(m[1, 2])   # 6  – рядок 1, стовпець 2
print(m[0])      # [1 2 3]  – весь перший рядок
print(m[:, 1])   # [2 5 8]  – весь другий стовпець
print(m[0:2, 1:3])
# [[2 3]
#  [5 6]]  – підматриця 2×2

# ── Булеве індексування ──
a = np.array([3, -1, 7, -4, 5, -2])
mask = a > 0
print(mask)      # [ True False  True False  True False]
```

```
print(a[mask])   # [3 7 5]  - тільки додатні елементи

# Скорочений запис
print(a[a > 0])  # [3 7 5]
print(a[a % 2 == 0])  # [-4 -2]  - тільки парні

# ── Зміна форми масиву ──
a = np.arange(12)       # [ 0  1  2 ... 11]
m = a.reshape(3, 4)     # матриця 3×4
v = m.flatten()         # назад у 1D
t = m.reshape(-1)       # теж у 1D (-1 = автоматично)

print(m)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# Транспонування матриці
print(m.T)              # рядки і стовпці міняються місцями
print(m.T.shape)        # (4, 3)
```

Арифметичні операції. Векторизація.

Головна перевага NumPy – операції над масивами виконуються поелементно без явних циклів. Це називається векторизацією і дозволяє писати компактний та швидкий код.

```
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

# ── Поелементні операції ──
print(a + b)      # [11 22 33 44]
print(a * b)      # [10 40 90 160]
print(b / a)      # [10. 10. 10. 10.]
print(a ** 2)     # [ 1  4  9 16]

# ── Broadcasting: операція масиву зі скаляром ──
# (scalar застосовується до КОЖНОГО елемента автоматично)
print(a * 3)      # [ 3  6  9 12]
print(a + 100)    # [101 102 103 104]
print(a > 2)      # [False False  True  True]

# ── Математичні функції (поелементні) ──
x = np.linspace(0, np.pi, 5)
print(np.sin(x))  # [0.    0.707 1.    0.707 0.   ]  (приблизно)
print(np.cos(x))
print(np.exp(x))  # e^x для кожного елемента
print(np.sqrt(np.array([1, 4, 9, 16])))  # [1. 2. 3. 4.]
print(np.log(np.array([1, np.e, np.e**2])))  # [0. 1. 2.]
print(np.abs(np.array([-3, 1, -5, 2])))  # [3 1 5 2]

# ── Агрегуючі функції ──
data = np.array([4, 7, 2, 9, 1, 5, 8, 3, 6])
print(data.sum())      # 45
print(data.mean())     # 5.0  - середнє
print(data.std())      # стандартне відхилення
print(data.min())      # 1
print(data.max())      # 9
```

```
print(data.argmin())   # 4  - ІНДЕКС мінімального
print(data.argmax())   # 3  - ІНДЕКС максимального
print(np.median(data)) # 5.0  - медіана

# — Агрегація по осях матриці —
m = np.array([[1, 2, 3],
              [4, 5, 6]])
print(m.sum(axis=0))  # [5 7 9]   - сума по стовпцях
print(m.sum(axis=1))  # [6 15]    - сума по рядках
print(m.mean(axis=1)) # [2. 5.]   - середнє по рядках
```

Лінійна алгебра (`np.linalg`). NumPy має вбудований підмодуль `linalg` (linear algebra) для розв'язання задач лінійної алгебри, що вивчаються у курсах вищої математики та фізики.

```
import numpy as np

# — Множення матриць —
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print(A @ B)        # матричне множення (оператор @)
# [[19 22]
#  [43 50]]

print(np.dot(A, B)) # те саме через np.dot()

# — Транспонування —
print(A.T)
# [[1 3]
#  [2 4]]

# — Визначник матриці —
print(np.linalg.det(A))   # -2.0

# — Обернена матриця —
A_inv = np.linalg.inv(A)
print(A_inv)
# [[-2.   1. ]
#  [ 1.5 -0.5]]

# Перевірка: A * A^(-1) = I (одинична матриця)
print(np.round(A @ A_inv, 10))
# [[1. 0.]
#  [0. 1.]]

# — Розв'язання системи лінійних рівнянь —
# 2x + 3y = 13
# x  - 2y = -4
A_sys = np.array([[2, 3],
                  [1, -2]])
b_vec = np.array([13, -4])

x_sol = np.linalg.solve(A_sys, b_vec)
print(x_sol)         # [2. 3.]  → x=2, y=3

# — Власні значення та власні вектори —
M = np.array([[4, 1], [2, 3]])
eigenvalues, eigenvectors = np.linalg.eig(M)
print('Власні значення:', eigenvalues)   # [5. 2.]
```

```
print('Власні вектори:\n', eigenvectors)

# ── Норма вектора ──
v = np.array([3, 4])
print(np.linalg.norm(v))  # 5.0  – довжина вектора
```

Статистика та обробка даних

```
import numpy as np

# Змоделюємо дані: оцінки 30 студентів
np.random.seed(42)   # фіксуємо генератор для відтворюваності
grades = np.random.randint(50, 101, size=30)
print(grades)

# ── Описова статистика ──
print(f'Середнє:     {grades.mean():.2f}')
print(f'Медіана:     {np.median(grades):.1f}')
print(f'Std:         {grades.std():.2f}')
print(f'Дисперсія:   {grades.var():.2f}')
print(f'Мін/Макс:    {grades.min()} / {grades.max()}')
print(f'Перцентиль 25: {np.percentile(grades, 25)}')
print(f'Перцентиль 75: {np.percentile(grades, 75)}')

# ── Сортування ──
sorted_grades = np.sort(grades)       # новий відсортований масив
grades.sort()                         # сортує масив на місці
print(np.argsort(grades))             # індекси відсортованих елементів

# ── Підрахунок унікальних значень ──
categories = np.array(['A','B','A','C','B','A','C','C'])
values, counts = np.unique(categories, return_counts=True)
for v, c in zip(values, counts):
    print(f'{v}: {c} разів')

# ── Коефіцієнт кореляції ──
height = np.array([165, 170, 175, 180, 185])
weight = np.array( [55,  65,  72,  80,  88])
corr = np.corrcoef(height, weight)
print(f'Кореляція зріст-вага: {corr[0,1]:.4f}')

# ── Апроксимація поліномом (метод найменших квадратів) ──
x = np.array([1, 2, 3, 4, 5])
y = np.array([2.1, 3.9, 6.2, 7.8, 10.1])
coeffs = np.polyfit(x, y, deg=1)   # лінійна апроксимація
print(f'y ≈ {coeffs[0]:.2f}x + {coeffs[1]:.2f}')

poly_func = np.poly1d(coeffs)      # об'єкт полінома
print(poly_func(3.5))              # значення y при x=3.5
```

Практичні інженерні та фізичні задачі з NumPy.
Розглянемо кілька типових задач, що зустрічаються у курсах фізики та інженерних дисциплін.

Приклад 1: Рух тіла, кинутого під кутом

```python
import numpy as np

# Параметри
v0    = 20.0          # початкова швидкість, м/с
angle = 45.0          # кут кидання, градуси
g     = 9.81          # прискорення вільного падіння

# Переводимо кут у радіани
theta = np.radians(angle)

# Проекції початкової швидкості
vx = v0 * np.cos(theta)
vy = v0 * np.sin(theta)

# Час польоту
T = 2 * vy / g

# Масив моментів часу: 200 точок від 0 до T
t = np.linspace(0, T, 200)

# Координати у кожен момент часу (векторизовано!)
x = vx * t
y = vy * t - 0.5 * g * t**2

# Дальність і максимальна висота
print(f'Дальність:       {x[-1]:.2f} м')
print(f'Макс. висота:    {y.max():.2f} м')
print(f'Час польоту:     {T:.2f} с')
# x та y — готові масиви для побудови графіку траєкторії
```

Приклад 2: Розв'язання системи рівнянь (закон Кірхгофа)

```python
import numpy as np

# Система рівнянь для трьох вузлів електричного кола:
# R1*I1 + R2*(I1-I2) = U1
# R3*I2 + R2*(I2-I1) = 0
# R4*(I1+I2) = U2

# Спрощена модель: 3 рівняння, 3 невідомих
# 5*x + 2*y - z  = 10
# 2*x + 8*y + 3*z = 20
#   x + y + 10*z = 15

R = np.array([[5,  2, -1],
              [2,  8,  3],
              [1,  1, 10]])

U = np.array([10, 20, 15])

I = np.linalg.solve(R, U)
print('Струми (A):')
for i, val in enumerate(I, 1):
```

```
    print(f'  I{i} = {val:.4f} A')

# Перевірка: R @ I повинно дорівнювати U
print('Перевірка:', np.allclose(R @ I, U))  # True
```

## Частина ІІ. Бібліотека Matplotlib.

`Matplotlib` – найпоширеніша бібліотека для побудови графіків у Python. Основний модуль – `pyplot` (зазвичай імпортується як plt).

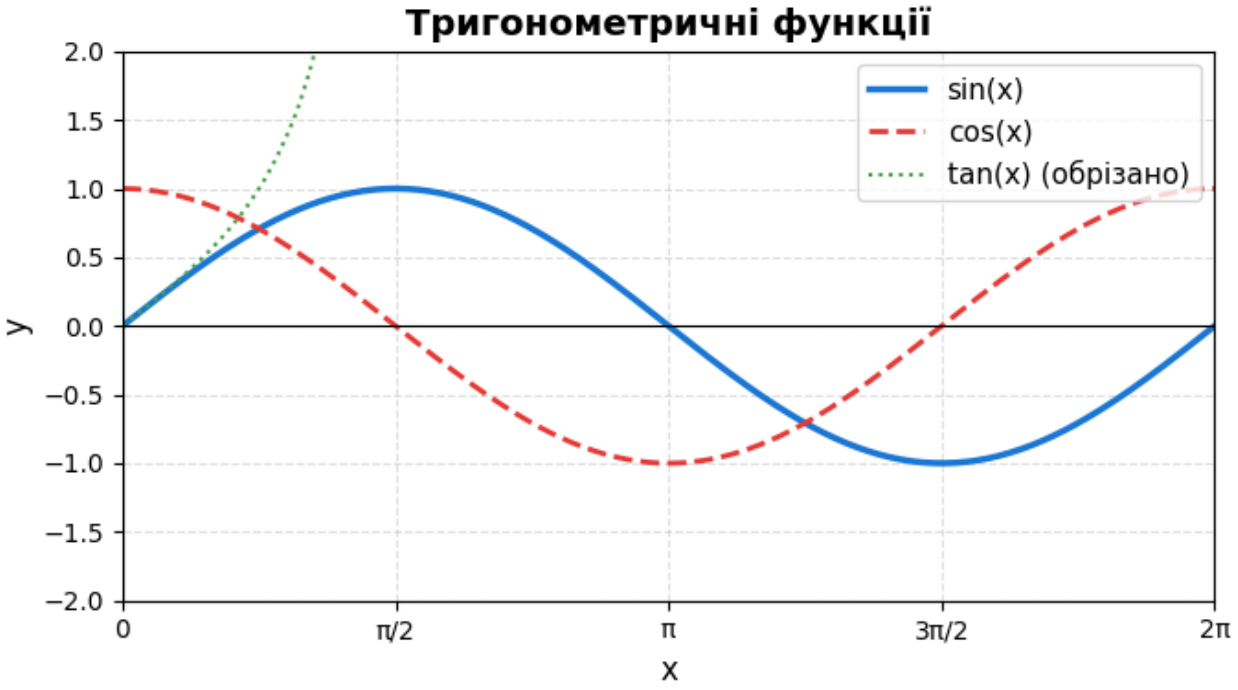
Два рівні роботи з Matplotlib:
- pyplot API (простий): `plt.plot()`, `plt.show()` – зручний для швидкої побудови
- Object-oriented API (гнучкий): `fig, ax = plt.subplots()` – рекомендований для складних графіків

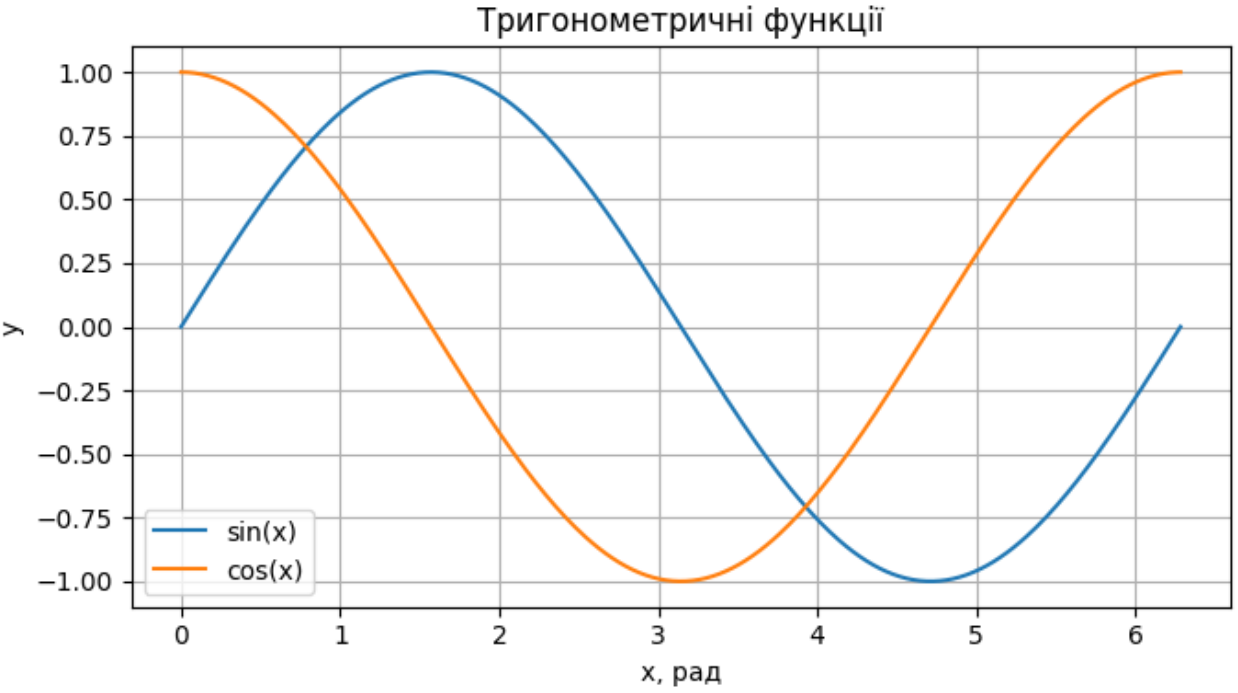
| Синтаксис | Опис / Результат |
|---|---|
| `plt.figure()` | Створює нове полотно (Figure). Параметр `figsize=(w, h)` задає розміри у дюймах. |
| `plt.subplots()` | Створює Figure та один або масив Axes (підграфіків). |
| `ax.plot(x, y)` | Лінійний графік. Параметри: color, linewidth, linestyle, label. |
| `ax.set_title(s)` | Заголовок графіка. |
| `ax.set_xlabel(s)` | Підпис осі X. |
| `ax.set_ylabel(s)` | Підпис осі Y. |
| `ax.legend()` | Відображає легенду (для кривих із label=). |
| `ax.grid(True)` | Відображає сітку. |
| `ax.set_xlim(a,b)` | Встановлює межі осі X. |
| `ax.set_ylim(a,b)` | Встановлює межі осі Y. |
| `plt.savefig(path)` | Зберігає рисунок у файл. Формат визначається розширенням. |
| `plt.show()` | Відображає графік у вікні (не потрібний при savefig у скриптах). |

Лінійні графіки. Форматування

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 300)

# ── Базовий графік ──
fig, ax = plt.subplots(figsize=(7, 4))

ax.plot(x, np.sin(x), label='sin(x)')
ax.plot(x, np.cos(x), label='cos(x)')

ax.set_title('Тригонометричні функції')
ax.set_xlabel('x, рад')
ax.set_ylabel('y')
ax.legend()
ax.grid(True)
plt.tight_layout()
plt.savefig('trig.png', dpi=150)
plt.show()

# ── Детальне форматування кривих ──
```

```python
fig, ax = plt.subplots(figsize=(7, 4))

ax.plot(x, np.sin(x),
        color='#1976D2',        # колір (hex)
        linewidth=2.5,          # товщина лінії
        linestyle='-',          # суцільна лінія
        label='sin(x)')

ax.plot(x, np.cos(x),
        color='#E53935',
        linewidth=2,
        linestyle='--',         # пунктирна
        label='cos(x)')

ax.plot(x, np.tan(np.clip(x, -1.4, 1.4)),
        color='#43A047',
        linewidth=1.5,
        linestyle=':',          # крапкова
        label='tan(x) (обрізано)')

# Форматування осей і підписів
ax.set_xlim(0, 2*np.pi)
ax.set_ylim(-2, 2)
ax.set_xticks([0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi])
ax.set_xticklabels(['0', 'π/2', 'π', '3π/2', '2π'])
ax.set_title('Тригонометричні функції', fontsize=14, fontweight='bold')
ax.set_xlabel('x', fontsize=12)
ax.set_ylabel('y', fontsize=12)
ax.legend(fontsize=11, loc='upper right')
ax.grid(True, alpha=0.4, linestyle='--')
ax.axhline(0, color='black', linewidth=0.8)  # вісь X
ax.axvline(0, color='black', linewidth=0.8)  # вісь Y

plt.tight_layout()
plt.savefig('trig_styled.png', dpi=150, bbox_inches='tight')
plt.show()
```


Тригонометричні функції
sin(x)
cos(x)
tan(x) (обрізано)
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
0
π/2
π
3π/2
2π
x


Тригонометричні функції
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
y
sin(x)
cos(x)
0
1
2
3
4
5
6
x, рад

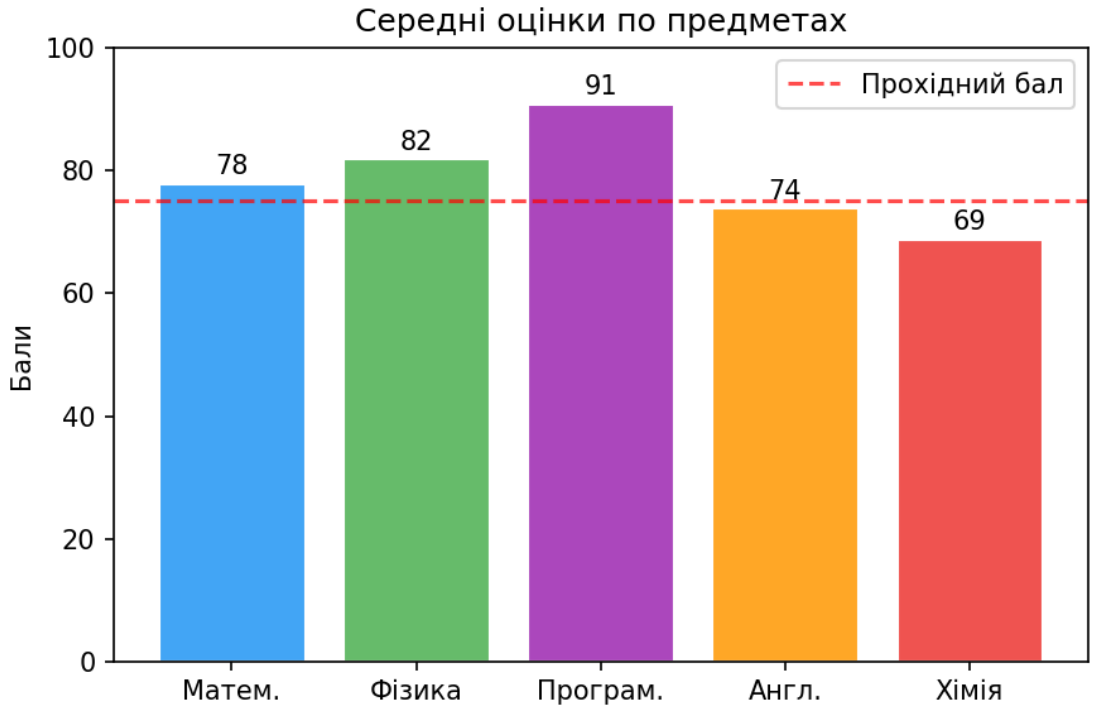
Виділення зон та фрагментів графіка. Matplotlib дозволяє підсвічувати окремі ділянки графіка – це дуже корисно для виділення критичних зон, допусків, фаз руху тощо.

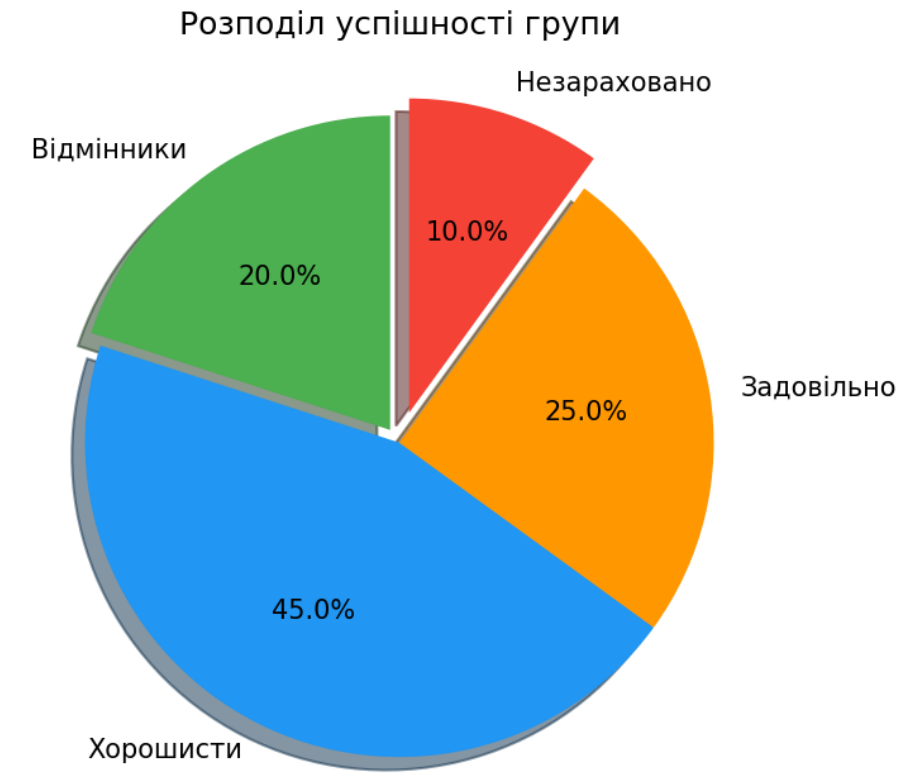
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 4 * np.pi, 500)
y = np.sin(x) * np.exp(-x * 0.15)   # загасаючі коливання

fig, ax = plt.subplots(figsize=(8, 4))

# Основна крива
ax.plot(x, y, color='#1565C0', linewidth=2, label='x(t) – загасання')

# ── Виділення небезпечної зони (axhspan) ──
# Горизонтальний span: виділяє зону між y=-0.3 і y=-1
ax.axhspan(-1.0, -0.3, alpha=0.15, color='red',
           label='Небезпечна зона')
ax.axhspan(0.3,   1.0, alpha=0.15, color='green',
           label='Допустима зона')

# ── Виділення фази часу (axvspan) ──
# Вертикальний span: виділяє часовий проміжок
ax.axvspan(0, np.pi, alpha=0.08, color='orange',
          label='Перша фаза')

# ── Горизонтальна лінія допуску ──
ax.axhline(0, color='gray', linewidth=0.8, linestyle='-')
ax.axhline( 0.3, color='green', linewidth=1, linestyle='--', alpha=0.7)
ax.axhline(-0.3, color='red',   linewidth=1, linestyle='--', alpha=0.7)

# ── Позначення конкретної точки ──
i_max = np.argmax(y)
ax.annotate(f'Макс: {y[i_max]:.2f}',
            xy=(x[i_max], y[i_max]),
            xytext=(x[i_max]+0.5, y[i_max]+0.2),
            arrowprops=dict(arrowstyle='->', color='black'),
            fontsize=9)

ax.set_title('Загасаючі коливання з виділенням зон', fontsize=13)
ax.set_xlabel('Час t, с')
ax.set_ylabel('Амплітуда x(t)')
ax.legend(fontsize=9, loc='upper right')
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('damping_zones.png', dpi=150)
plt.show()
```

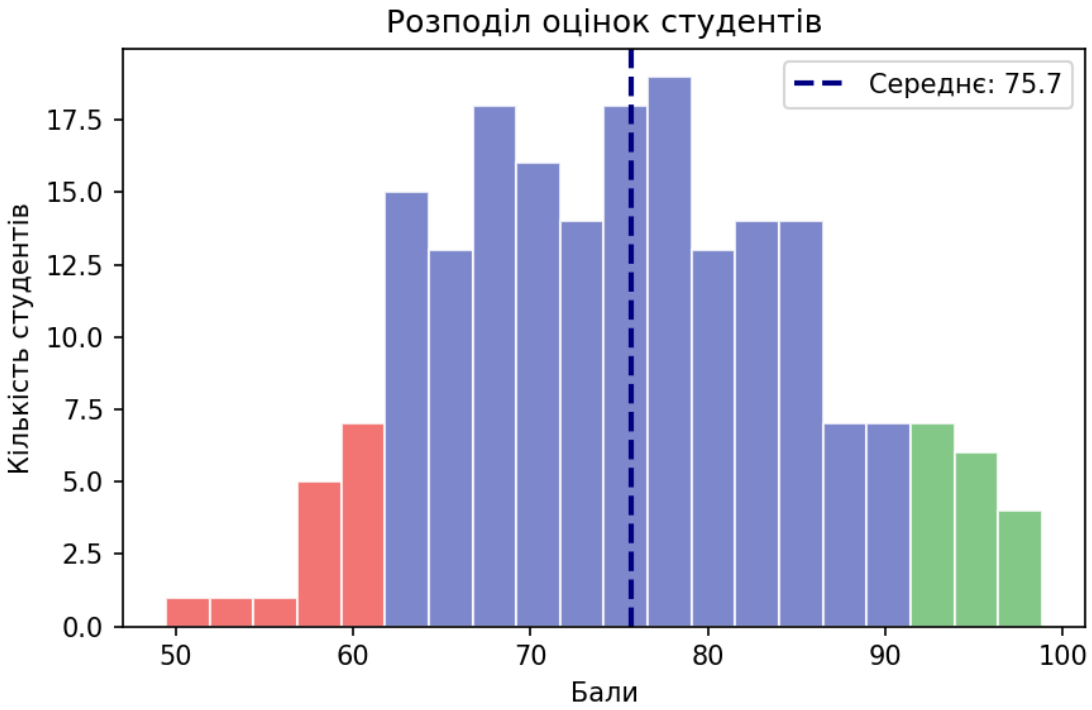

Різні типи графіків

```python
import numpy as np
import matplotlib.pyplot as plt

# ── Стовпчаста діаграма (bar) ──
subjects = ['Матем.', 'Фізика', 'Програм.', 'Англ.', 'Хімія']
avg_scores = [78, 82, 91, 74, 69]
colors = ['#42A5F5','#66BB6A','#AB47BC','#FFA726','#EF5350']

fig, ax = plt.subplots(figsize=(6, 4))
bars = ax.bar(subjects, avg_scores, color=colors, edgecolor='white',
              linewidth=1.5)

# Підписи значень над стовпцями
for bar, val in zip(bars, avg_scores):
    ax.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.5,
            str(val), ha='center', va='bottom', fontsize=10)

ax.set_ylim(0, 100)
ax.set_title('Середні оцінки по предметах')
ax.set_ylabel('Бали')
ax.axhline(75, color='red', linestyle='--', alpha=0.7, label='Прохідний бал')
ax.legend()
plt.tight_layout()
plt.savefig('bar_chart.png', dpi=150)
plt.show()

# ── Кругова діаграма (pie) ──
categories = ['Відмінники', 'Хорошисти', 'Задовільно', 'Незараховано']
sizes = [20, 45, 25, 10]
explode = (0.05, 0, 0, 0.1)   # виділити скибки
pie_colors = ['#4CAF50','#2196F3','#FF9800','#F44336']

fig, ax = plt.subplots(figsize=(5, 5))
ax.pie(sizes, labels=categories, explode=explode,
       colors=pie_colors, autopct='%1.1f%%',
       startangle=90, shadow=True)
```

```
ax.set_title('Розподіл успішності групи')
plt.tight_layout()
plt.savefig('pie_chart.png', dpi=150)
plt.show()

# ── Гістограма (hist) ──
np.random.seed(0)
data = np.random.normal(75, 10, 200)  # нормальний розподіл

fig, ax = plt.subplots(figsize=(6, 4))
n, bins, patches = ax.hist(data, bins=20, color='#5C6BC0',
                           edgecolor='white', alpha=0.8)

# Виділити кольором різні діапазони
for patch, left in zip(patches, bins[:-1]):
    if left < 60:
        patch.set_facecolor('#EF5350')
    elif left >= 90:
        patch.set_facecolor('#66BB6A')

ax.axvline(data.mean(), color='navy', linewidth=2,
           linestyle='--', label=f'Середнє: {data.mean():.1f}')
ax.set_title('Розподіл оцінок студентів')
ax.set_xlabel('Бали')
ax.set_ylabel('Кількість студентів')
ax.legend()
plt.tight_layout()
plt.savefig('histogram.png', dpi=150)
plt.show()
```



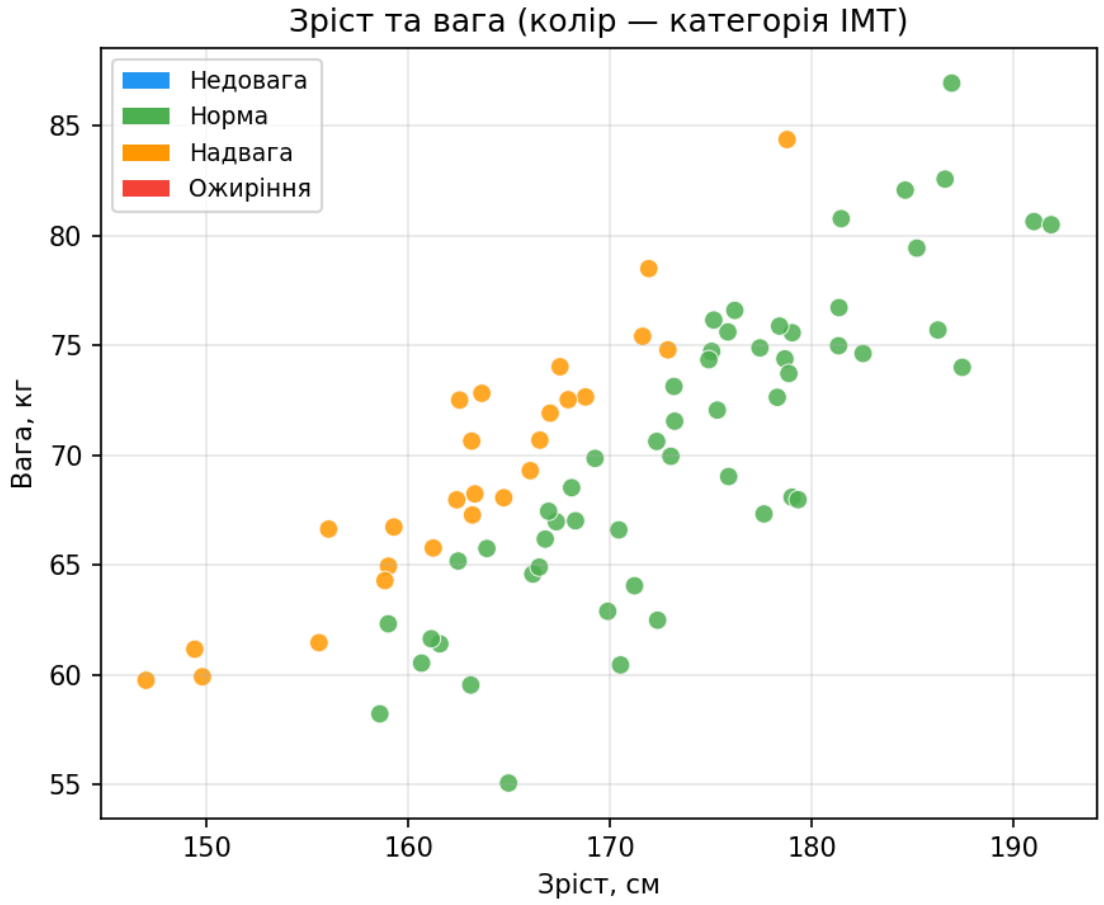

Розподіл успішності групи
Незараховано
Відмінники
10.0%
20.0%
25.0%
Задовільно
45.0%
Хорошисти

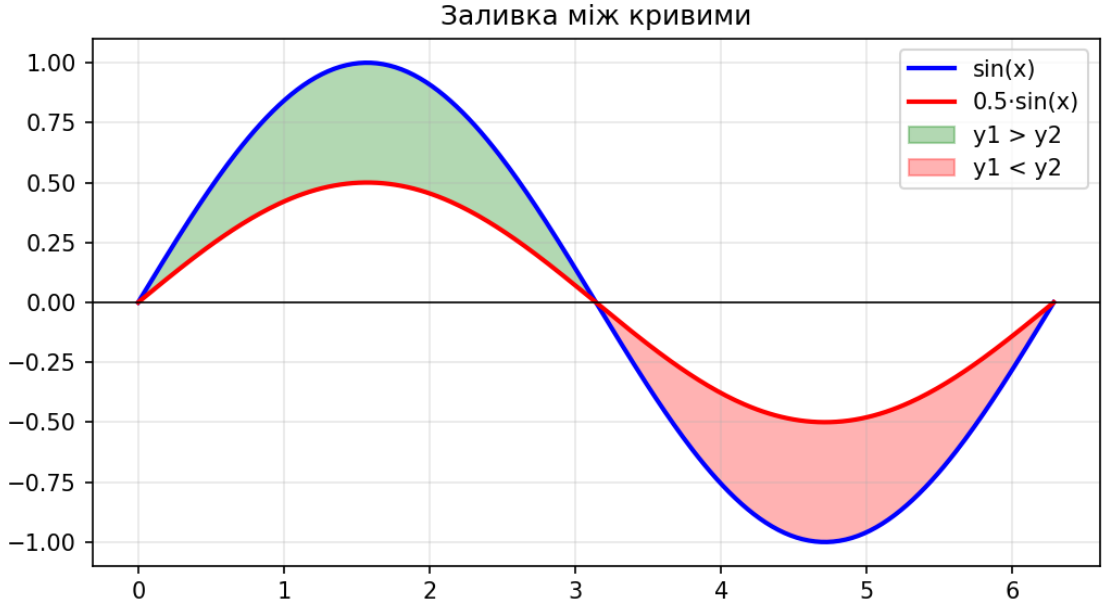

Розподіл оцінок студентів
Середнє: 75.7
Кількість студентів
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
50
60
70
80
90
100
Бали

Діаграма розсіювання. Заливка між кривими

```
import numpy as np
import matplotlib.pyplot as plt

# ── Scatter (точкова діаграма) ──
np.random.seed(1)
n = 80
height = np.random.normal(170, 10, n)
weight = height * 0.45 - 7 + np.random.normal(0, 5, n)
bmi    = weight / (height/100)**2

# Колір точки залежить від ІМТ
colors = np.where(bmi < 18.5, '#2196F3',
         np.where(bmi < 25,   '#4CAF50',
         np.where(bmi < 30,   '#FF9800', '#F44336')))

fig, ax = plt.subplots(figsize=(6, 5))
scatter = ax.scatter(height, weight, c=colors, s=50,
                     edgecolors='white', linewidths=0.5, alpha=0.85)

# Ручна легенда
from matplotlib.patches import Patch
legend_elements = [
    Patch(facecolor='#2196F3', label='Недовага'),
    Patch(facecolor='#4CAF50', label='Норма'),
    Patch(facecolor='#FF9800', label='Надвага'),
    Patch(facecolor='#F44336', label='Ожиріння'),
]
ax.legend(handles=legend_elements, fontsize=9)
ax.set_title('Зріст та вага (колір – категорія ІМТ)')
ax.set_xlabel('Зріст, см')
ax.set_ylabel('Вага, кг')
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('scatter_bmi.png', dpi=150)
plt.show()

# ── fill_between – заливка між кривими ──
x = np.linspace(0, 2*np.pi, 300)
y1 = np.sin(x)
y2 = np.sin(x) * 0.5

fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(x, y1, 'b-', linewidth=2, label='sin(x)')
ax.plot(x, y2, 'r-', linewidth=2, label='0.5·sin(x)')

# Заливка між кривими (де y1 > y2 – зелено, де y1 < y2 – червоно)
ax.fill_between(x, y1, y2,
                where=(y1 >= y2), alpha=0.3, color='green',
                label='y1 > y2')
ax.fill_between(x, y1, y2,
                where=(y1 < y2),  alpha=0.3, color='red',
                label='y1 < y2')

ax.axhline(0, color='black', linewidth=0.8)
ax.set_title('Заливка між кривими')
ax.legend()
```

```
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('fill_between.png', dpi=150)
plt.show()
```

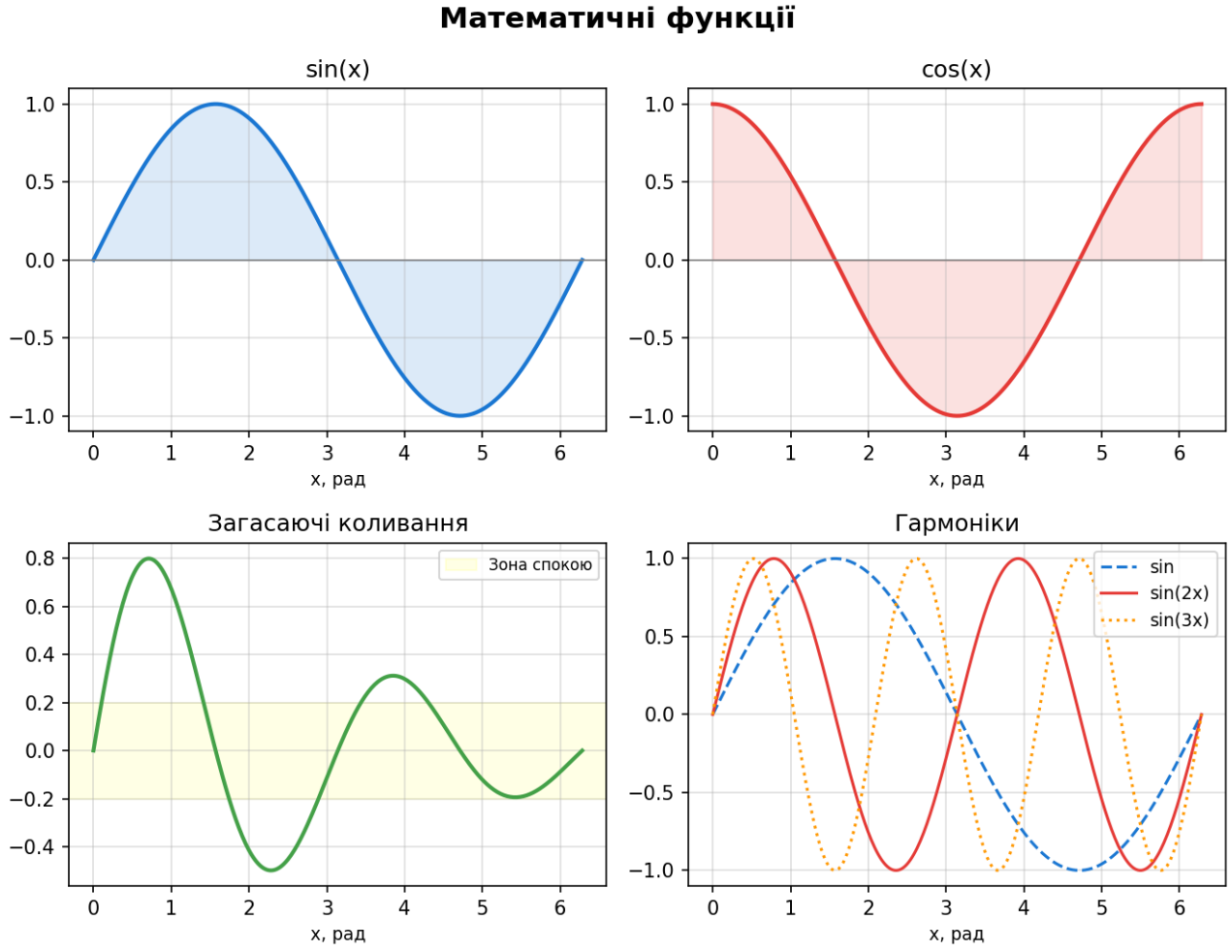


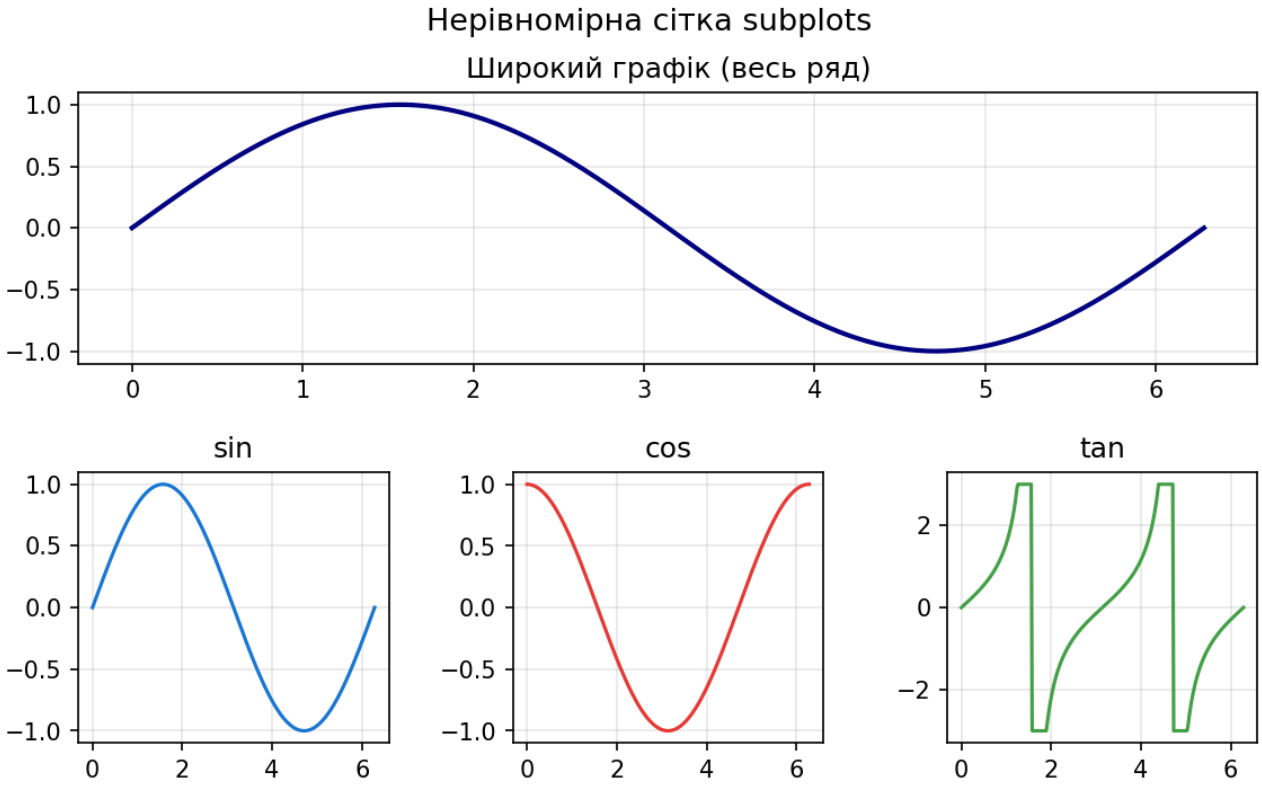



Subplots: кілька графіків на одному полотні

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 300)

# ── 2×2 сітка підграфіків ──
fig, axes = plt.subplots(2, 2, figsize=(9, 7))
fig.suptitle('Математичні функції', fontsize=15, fontweight='bold')

# Підграфік [0,0] – sin(x)
axes[0, 0].plot(x, np.sin(x), color='#1976D2', linewidth=2)
axes[0, 0].fill_between(x, np.sin(x), 0, alpha=0.15, color='#1976D2')
axes[0, 0].set_title('sin(x)')
axes[0, 0].grid(True, alpha=0.4)
axes[0, 0].axhline(0, color='gray', linewidth=0.8)

# Підграфік [0,1] – cos(x)
axes[0, 1].plot(x, np.cos(x), color='#E53935', linewidth=2)
axes[0, 1].fill_between(x, np.cos(x), 0, alpha=0.15, color='#E53935')
axes[0, 1].set_title('cos(x)')
axes[0, 1].grid(True, alpha=0.4)
axes[0, 1].axhline(0, color='gray', linewidth=0.8)

# Підграфік [1,0] – exp(-x)*sin(x) загасаючі коливання
y_damp = np.exp(-x*0.3) * np.sin(2*x)
axes[1, 0].plot(x, y_damp, color='#43A047', linewidth=2)
axes[1, 0].axhspan(-0.2, 0.2, alpha=0.1, color='yellow',
                   label='Зона спокою')
axes[1, 0].set_title('Загасаючі коливання')
axes[1, 0].legend(fontsize=8)
axes[1, 0].grid(True, alpha=0.4)
# Підграфік [1,1] – кілька функцій разом
axes[1, 1].plot(x, np.sin(x),   '--', color='#1976D2', label='sin')
axes[1, 1].plot(x, np.sin(2*x), '-',  color='#E53935', label='sin(2x)')
axes[1, 1].plot(x, np.sin(3*x), ':',  color='#FF9800', label='sin(3x)')
axes[1, 1].set_title('Гармоніки')
axes[1, 1].legend(fontsize=9)
axes[1, 1].grid(True, alpha=0.4)

# Спільний підпис осей
for ax in axes.flat:
    ax.set_xlabel('x, рад', fontsize=9)

plt.tight_layout()
plt.savefig('subplots_2x2.png', dpi=150, bbox_inches='tight')
plt.show()

# ── Нерівномірна сітка через gridspec ──
fig = plt.figure(figsize=(9, 5))
gs  = fig.add_gridspec(2, 3, hspace=0.4, wspace=0.4)

ax_wide = fig.add_subplot(gs[0, :])   # верхній рядок – весь ряд
ax_bl   = fig.add_subplot(gs[1, 0])
ax_bm   = fig.add_subplot(gs[1, 1])
ax_br   = fig.add_subplot(gs[1, 2])
```

```
ax_wide.plot(x, np.sin(x), color='navy', linewidth=2)
ax_wide.set_title('Широкий графік (весь ряд)')
ax_wide.grid(True, alpha=0.3)

for ax, func, clr, lbl in [
    (ax_bl, np.sin, '#1976D2', 'sin'),
    (ax_bm, np.cos, '#E53935', 'cos'),
    (ax_br, np.tan, '#43A047', 'tan'),
]:
    ax.plot(x, np.clip(func(x),-3,3), color=clr, linewidth=1.5)
    ax.set_title(lbl)
    ax.grid(True, alpha=0.3)

fig.suptitle('Нерівномірна сітка subplots', fontsize=13)
plt.savefig('gridspec.png', dpi=150, bbox_inches='tight')
plt.show()
```



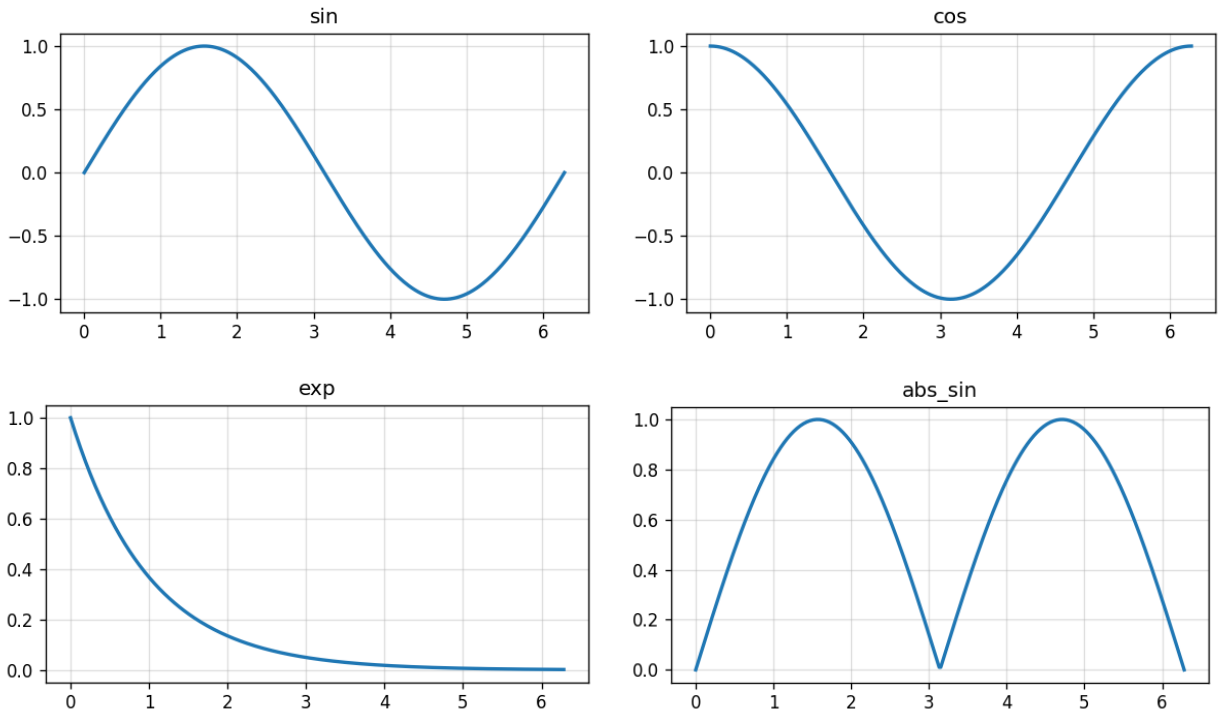



Збереження графіків у файл. Функція `plt.savefig()` зберігає поточний графік у файл. Формат визначається розширенням імені файлу.

| Синтаксис | Опис / Результат |
|---|---|
| `.png` | PNG – растровий, з прозорістю. Найбільш поширений для звітів. |
| `.pdf` | PDF – векторний, масштабується без втрат. Ідеальний для публікацій. |
| `.svg` | SVG – векторний XML. Можна редагувати у Inkscape/Illustrator. |
| `.eps` | EPS – векторний, для LaTeX та наукових журналів. |
| `.jpg` | JPEG – стиснений, без прозорості. Менший розмір, але артефакти. |

```python
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2*np.pi, 300)
fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(x, np.sin(x), linewidth=2)
ax.set_title('Приклад збереження')

# Основні параметри savefig:
# dpi    - роздільна здатність (72 - екран, 150-300 - друк)
# bbox_inches='tight' - обрізати зайві поля
# transparent=True    - прозорий фон (для PNG)
# facecolor           - колір фону рисунку

plt.savefig('graph.png',  dpi=150, bbox_inches='tight')
plt.savefig('graph.pdf',  bbox_inches='tight')
plt.savefig('graph.svg',  bbox_inches='tight')
plt.savefig('graph_transparent.png', dpi=150,
            transparent=True, bbox_inches='tight')
```

```python
# Збереження у визначену папку
from pathlib import Path
output_dir = Path('lab7_output')
output_dir.mkdir(exist_ok=True)

plt.savefig(output_dir / 'sine_wave.png', dpi=200)
print('Графік збережено!')

plt.show()

# Збереження кількох графіків у цикл
functions = {'sin': np.sin, 'cos': np.cos,
             'exp': lambda x: np.exp(-x),
             'abs_sin': lambda x: np.abs(np.sin(x))}

for name, func in functions.items():
    fig, ax = plt.subplots(figsize=(5, 3))
    ax.plot(x, func(x), linewidth=2)
    ax.set_title(name)
    ax.grid(True, alpha=0.4)
    plt.tight_layout()
    plt.savefig(output_dir / f'{name}.png', dpi=120)
    plt.close()   # ВАЖЛИВО: закривати після збереження у циклі!
    print(f'Збережено: {name}.png')
```




Завдання на лабораторну роботу;

Перед виконанням встановіть бібліотеки:

```
pip install numpy matplotlib scipy
```

Усі графіки зберігайте у папці `lab7_output/` у форматах PNG та PDF.

7.1. Масиви NumPy та базові операції. Виконайте такі дії з масивами:

- створіть масив `a` з 20 рівномірно розміщених чисел від 0 до 10 (`np.linspace`) та масив `b` з 20 цілих випадкових чисел від 1 до 50;
- виведіть: суму, добуток, різницю масивів; масив `a` у квадраті; масив `b`, де всі непарні елементи замінено на 0 (булеве індексування);
- знайдіть індекси елементів масиву `b`, що перевищують 30, виведіть самі ці елементи;
- перетворіть `b` у матрицю 4×5, виведіть суму по рядках і стовпцях; транспонуйте матрицю.

7.2. Тригонометричні функції та їх графіки. Побудуйте на одному полотні (subplots 2×1) два графіки:

- верхній: `sin(x)`, `cos(x)` і `sin(x)+cos(x)` на проміжку `[0, 4π]` з різними кольорами, стилями ліній і легендою;
- нижній: `sin²(x)` і `cos²(x)`, а між ними – заливка `fill_between`; поясніть словами (у `title` або `annotation`), чому `sin²(x)+cos²(x)=1`;
- Виділіть осьовими лініями `axhline/axvline` нулі, позначте максимуми кривих маркерами

Збережіть у trig_lab.png та trig_lab.pdf

7.3. Статистика і гістограма. Згенеруйте масив із 500 значень нормального розподілу N(70, 12) – імітація оцінок великого курсу. Побудуйте на одному полотні (1×2):

- зліва: гістограму з 25 стовпцями; виділіть червоним стовпці до 60 балів, зеленим – від 90; нанесіть вертикальні лінії середнього та медіани;
- справа: кругову діаграму розподілу по категоріях (< 60, 60–74, 75–89, ≥ 90).

Виведіть у консоль описову статистику: середнє, медіана, std, min, max, перцентилі 10 і 90.

7.4. Фізика: рух тіла, кинутого під кутом. На основі прикладу з теоретичної частини (розділ 3.6) побудуйте інтерактивне дослідження:

- для кутів 30°, 45°, 60° і 75° за однакової початкової швидкості $v_0$ = 25 м/с обчисліть траєкторії (масиви x та y через NumPy);
- побудуйте всі траєкторії на одному графіку різними кольорами, у легенді вкажіть кут і дальність польоту;
- виділіть зону `axhspan` від 0 до 5 м – «зона перешкоди»; побудуйте аннотацію для максимальної висоти кожної траєкторії;
- на окремому підграфіку (subplots 1×2) покажіть залежність дальності польоту від кута (від 0° до 90° через 1°) та залежність максимальної висоти від кута.

Збережіть як projectile_motion.png і projectile_motion.pdf.

7.5. Лінійна алгебра: системи рівнянь та матриці. Виконайте такі задачі з np.linalg:

- задайте матрицю A 3×3 з довільними ненульовими числами. Обчисліть: det(A), $A^{-1}$, $A \cdot A^{-1}$ (перевірка = одинична). Виведіть результати з поясненнями;
- розв'яжіть систему трьох лінійних рівнянь (придумайте власну або використайте з курсу вищої математики). Перевірте правильність через підстановку;
- знайдіть власні значення та власні вектори матриці. Побудуйте на графіку (стрілки quiver або plot) початкові та трансформовані вектори – наочно покажіть, як матриця розтягує/повертає вектори.

## Контрольні запитання

1. Що таке масив `ndarray` у NumPy? Чим він відрізняється від звичайного списку Python за структурою зберігання та швидкістю обчислень?
2. Що таке векторизація у NumPy? Поясніть на прикладі, чому 'a * 3' для NumPy-масиву швидше, ніж цикл for.
3. Що означають атрибути `shape`, `ndim`, `size` та `dtype` масиву? Що поверне вираз `np.zeros((3,4)).shape`?
4. Чим відрізняються `np.arange()` та `np.linspace()`? Коли доцільно використовувати кожну з них?
5. Поясніть принцип булевого індексування. Що поверне вираз `a[a > 5]` для масиву `a = np.array([3, 7, 1, 9, 4])`?
6. Що таке `broadcasting` у NumPy? Як виконується операція між масивом форми (3,4) і скаляром?
7. Для чого використовується `np.linalg.solve()`? Запишіть, яку задачу розв'язує цей виклик: `np.linalg.solve(A, b)`.
8. Що таке `Figure` та `Axes` у Matplotlib? Поясніть різницю між `plt.plot()` та `ax.plot()`.
9. Для чого використовуються функції `axhspan()` та `axvspan()`? Наведіть практичний приклад їхнього застосування.
10. Назвіть чотири формати збереження графіків у Matplotlib. Які переваги PNG перед PDF і навпаки?

Лабораторна робота №15.

**Обробка та аналіз даних з бібліотекою Pandas.**

**Мета роботи:** Зрозуміти поняття аналізу даних та їхню роль у прийнятті рішень; ознайомитися з основними структурами Pandas – DataFrame та Series – та навчитися завантажувати дані з різних форматів файлів (CSV, Excel, JSON).
Опанувати ключові техніки попередньої обробки даних: виявлення та заповнення пропусків, видалення дублікатів, зміна типів даних, фільтрація рядків та вибірка стовпців; навчитися застосовувати функції до даних через apply та vectorized-операції.
Засвоїти техніки агрегації та групування даних через groupby, злиття таблиць через merge та concat, а також навчитися будувати базові аналітичні графіки засобами Pandas і Matplotlib для візуального представлення результатів аналізу.

## Теоретичні відомості:

### Вступ: Навіщо аналізувати дані? Що таке аналіз даних і чому він важливий

Щодня у світі генерується близько 2.5 квінтильйонів байт даних. Смартфони фіксують маршрути, банки зберігають транзакції, лікарні накопичують медичні картки, університети ведуть успішність студентів. Але самі по собі ці дані – просто числа та рядки. Цінність виникає тоді, коли їх починають аналізувати.
Аналіз даних – це процес вивчення набору даних з метою виявлення закономірностей, перевірки гіпотез та отримання корисних висновків для прийняття рішень. Простіше кажучи: ми перетворюємо «сирі» числа на корисне знання.

Ось кілька конкретних прикладів, як аналіз даних застосовується прямо зараз:

- Медицина: аналіз електронних медичних карток тисяч пацієнтів дозволяє виявити фактори ризику серцево-судинних захворювань задовго до появи симптомів
- Освіта: аналіз успішності студентів допомагає виявити тих, хто ризикує не скласти сесію, і вчасно запропонувати допомогу
- Транспорт: Uber аналізує мільйони поїздок, щоб оптимізувати ціни, маршрути водіїв і прогнозувати попит у різних районах міста
- Бізнес: Netflix аналізує, що і коли дивляться 200+ мільйонів підписників – і на основі цього вирішує, які серіали виробляти
- Екологія: аналіз даних метеостанцій і супутників дозволяє моделювати зміни клімату і прогнозувати природні катастрофи

### Цикл аналізу даних: від запитання до відповіді

Будь-який аналіз починається не з коду, а з запитання. «Чому впали продажі у жовтні?», «Які студенти найімовірніше відраховуються?», «Де найбільше пробок у місті?» Далі аналіз проходить через кілька етапів:

- Формулюємо конкретне питання. Погане питання: «Проаналізуй дані». Хороше: «Чи є залежність між кількістю годин сну студента і його GPA?»
- Знаходимо або генеруємо дані. Джерела: CSV-файли, бази даних, API, веб-скрейпінг, опитування, датчики.
- Реальні дані ніколи не бувають ідеальними. Це найчастіше найдовший етап (до 80% часу!): заповнення пропусків, виправлення помилок, видалення дублікатів, уніфікація

форматів.
- Вивчаємо дані: розподіли, кореляції, аномалії. Будуємо графіки, рахуємо статистику. Мета – «відчути» дані та сформулювати гіпотези.
- Перевіряємо гіпотези: порівнюємо групи, будуємо регресії, шукаємо закономірності.
- Перетворюємо технічні результати на зрозумілі відповіді для людей, що приймають рішення. Графік зрозуміліший за таблицю цифр.

***Ключовий принцип аналізу даних:***
*Кореляція НЕ означає причинно-наслідковий зв'язок!*
*Факт: у містах з більшою кількістю церков більше злочинів.*
*Причина: обидва показники ростуть зі збільшенням населення міста.*
*Завжди запитуй: «Чи може бути третій фактор, що пояснює цю залежність?»*

## Pandas у екосистемі Python для аналізу даних

Python став мовою №1 для аналізу даних завдяки своїй екосистемі бібліотек. Ось як вони взаємодіють:
- NumPy – фундамент: масиви ndarray, математичні операції. Pandas побудований поверх NumPy.
- Pandas – обробка табличних даних: завантаження, очищення, фільтрація, групування, злиття.
- Matplotlib / Seaborn – візуалізація: графіки, діаграми, теплові карти.
- Scikit-learn – машинне навчання: класифікація, регресія, кластеризація. Приймає дані у форматі Pandas DataFrame.
- Statsmodels – статистичне моделювання: регресійний аналіз, перевірка гіпотез.

У цій лабораторній роботі ми зосередимося на Pandas – центральному інструменті будь-якого аналізу даних у Python. Коли у майбутньому ви вивчатимете Data Science або Machine Learning, навички роботи з Pandas будуть необхідні на кожному кроці.

### Встановлення та основні структури Pandas

```
# Встановлення
pip install pandas openpyxl xlrd matplotlib seaborn

import pandas as pd
import numpy as np
print(pd.__version__)   # 2.x.x
```

### Series – одновимірна структура

Series – це пронумерована колекція значень з мітками (індексом). Схожа на стовпець таблиці або словник.

```
import pandas as pd
import numpy as np

# ── Створення Series ──
s1 = pd.Series([10, 20, 30, 40, 50])
print(s1)
# 0    10
# 1    20  ...
# dtype: int64
```

```python
# З власним індексом
grades = pd.Series(
    [4.2, 4.8, 3.5, 4.6],
    index=['Іван', 'Марія', 'Олег', 'Ліна'],
    name='GPA'
)
print(grades['Марія'])    # 4.8
print(grades[grades > 4]) # Іван 4.2, Марія 4.8, Ліна 4.6

# Атрибути
print(grades.dtype)    # float64
print(grades.index)    # Index(['Іван', 'Марія', 'Олег', 'Ліна'])
print(grades.values)   # [4.2 4.8 3.5 4.6]
print(len(grades))     # 4

# Операції над Series (векторизовані)
print(grades * 20)     # кожен елемент * 20
print(grades.mean())   # 4.275
print(grades.max())    # 4.8
```

**DataFrame – двовимірна структура (таблиця)**

DataFrame – це таблиця з рядками та стовпцями, де кожен стовпець – це Series з однаковим індексом. Аналог аркуша Excel або таблиці SQL.

```python
# ── Створення DataFrame ──

# Зі словника (ключ = назва стовпця)
df = pd.DataFrame({
    'name':   ['Іван', 'Марія', 'Олег', 'Ліна', 'Тарас'],
    'year':   [2, 3, 1, 2, 4],
    'gpa':    [4.2, 4.8, 3.5, 4.6, 3.9],
    'city':   ['Київ', 'Харків', 'Київ', 'Львів', 'Одеса'],
    'active': [True, True, False, True, True],
})

# ── Базова інформація ──
print(df.shape)        # (5, 5) – 5 рядків, 5 стовпців
print(df.dtypes)       # типи кожного стовпця
print(df.columns)      # Index(['name', 'year', 'gpa', 'city', 'active'])
print(df.index)        # RangeIndex(start=0, stop=5, step=1)

# ── Перегляд даних ──
print(df.head(3))      # перші 3 рядки
print(df.tail(2))      # останні 2 рядки
print(df.sample(2))    # 2 випадкові рядки

# ── Статистичний огляд ──
print(df.info())       # типи, пропуски, пам'ять
print(df.describe())   # count, mean, std, min, max, quartiles
# describe() для рядкових стовпців:
print(df.describe(include='object'))  # count, unique, top, freq
```

Читання та збереження даних

| Метод / Синтаксис | Опис / Результат |
|---|---|
| `pd.read_csv(path)` | Читає CSV-файл. Параметри: sep, encoding, header, index_col. |

| | |
|---|---|
| `pd.read_excel(path)` | Читає Excel (.xlsx). Параметр sheet_name. |
| `pd.read_json(path)` | Читає JSON-файл або рядок. |
| `pd.read_sql(query, conn)` | Виконує SQL-запит і повертає DataFrame. |
| `pd.read_clipboard()` | Читає дані з буфера обміну (зручно для швидкого тесту). |
| `df.to_csv(path)` | Зберігає у CSV. index=False – без стовпця індексів. |
| `df.to_excel(path)` | Зберігає у Excel. sheet_name – назва аркуша. |
| `df.to_json(path)` | Зберігає у JSON. orient='records' – список об'єктів. |
| `df.to_dict(orient)` | Перетворює на словник Python. |

```python
import pandas as pd

# ── Читання CSV ──
df = pd.read_csv('students.csv',
    sep=',',              # роздільник (може бути ';' для Excel-CSV)
    encoding='utf-8',     # кодування
    index_col='id',       # стовпець як індекс
    parse_dates=['enrolled_at'],  # автоматично парсить дату
    na_values=['N/A', '-', ''],   # що вважати NaN
)

# ── Читання Excel ──
df_excel = pd.read_excel('grades.xlsx',
    sheet_name='Sheet1',
    header=0,             # рядок з назвами стовпців
    skiprows=2,           # пропустити 2 рядки зверху
)

# ── Читання кількох аркушів ──
sheets = pd.read_excel('report.xlsx', sheet_name=None)  # dict
for name, df in sheets.items():
    print(f'Аркуш {name}: {df.shape}')

# ── Збереження ──
df.to_csv('output.csv', index=False, encoding='utf-8-sig')
df.to_excel('output.xlsx', sheet_name='Students', index=False)
df.to_json('output.json', orient='records', force_ascii=False, indent=2)

# ── Читання з URL ──
url = 'https://raw.githubusercontent.com/.../data.csv'
# df = pd.read_csv(url)   # Pandas вміє читати з HTTP

# ── Тестові дані прямо у коді (без файлу) ──
from io import StringIO
csv_data = '''name,year,gpa
Іван,2,4.2
Марія,3,4.8'''
df_test = pd.read_csv(StringIO(csv_data))
```

Вибірка та індексування даних

```python
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'name': ['Іван','Марія','Олег','Ліна','Тарас'],
```

```
    'year': [2, 3, 1, 2, 4],
    'gpa':  [4.2, 4.8, 3.5, 4.6, 3.9],
    'city': ['Київ','Харків','Київ','Львів','Одеса'],
})

# ── Вибір стовпців ──
print(df['name'])              # Series
print(df[['name', 'gpa']])     # DataFrame (подвійні дужки!)

# ── .loc[] – за МІТКАМИ індексу ──
print(df.loc[0])               # перший рядок (Series)
print(df.loc[0, 'gpa'])        # значення конкретної комірки
print(df.loc[1:3, 'name':'gpa'])  # зріз рядків і стовпців

# ── .iloc[] – за ПОЗИЦІЄЮ (числовий індекс) ──
print(df.iloc[0])              # перший рядок
print(df.iloc[0, 2])           # рядок 0, стовпець 2
print(df.iloc[:3, :2])         # перші 3 рядки, перші 2 стовпці
print(df.iloc[-1])             # останній рядок

# ── Булева вибірка (фільтрація) ──
good = df[df['gpa'] >= 4.5]
print(good)

# Комбінована умова: & (і), | (або), ~ (не)
kyiv_good = df[(df['city'] == 'Київ') & (df['gpa'] > 4.0)]
not_first = df[df['year'] != 1]

# .isin() – входження у список
big_cities = df[df['city'].isin(['Київ', 'Харків'])]

# .str методи для рядкових стовпців
ivan = df[df['name'].str.contains('ів', case=False)]
starts_M = df[df['name'].str.startswith('М')]

# .query() – більш читабельний синтаксис
result = df.query('gpa >= 4.0 and year in [2, 3]')
result2 = df.query('city == @target_city', engine='python')

# ── Встановлення значення ──
df.loc[df['gpa'] < 3.0, 'gpa'] = 3.0    # мін. GPA = 3.0
df.at[0, 'city'] = 'Дніпро'             # одне значення
```

## Очищення та перетворення даних

Реальні дані майже завжди «брудні»: пропуски, помилки, дублікати, неправильні типи. Очищення – найважливіший етап аналізу.

**Робота з пропущеними значеннями (NaN)**

```
import pandas as pd
import numpy as np

# Створимо «брудний» датасет
df = pd.DataFrame({
    'name': ['Іван', 'Марія', None, 'Ліна', 'Тарас'],
    'age':  [20, np.nan, 22, np.nan, 24],
    'gpa':  [4.2, 4.8, 3.5, np.nan, 3.9],
    'city': ['Київ', 'Харків', 'Київ', 'Львів', 'Київ'],
})
```

```
# ── Виявлення пропусків ──
print(df.isnull())              # булева маска
print(df.isnull().sum())        # кількість NaN у кожному стовпці
print(df.isnull().sum() / len(df) * 100)  # відсоток пропусків

# ── Видалення рядків/стовпців з пропусками ──
df_clean = df.dropna()          # видалити рядки де є ХОЧА Б ОДИН NaN
df_clean2 = df.dropna(how='all')    # видалити тільки якщо ВСІ NaN
df_clean3 = df.dropna(subset=['name', 'gpa'])  # тільки якщо NaN у цих стовпцях
df_clean4 = df.dropna(thresh=3)    # залишити рядки з мінімум 3 не-NaN

# ── Заповнення пропусків ──
df['age'].fillna(df['age'].mean(), inplace=True)    # середнє
df['gpa'].fillna(df['gpa'].median(), inplace=True)  # медіана
df['name'].fillna('Невідомо', inplace=True)         # константа

# forward fill: заповнити попереднім значенням
df['city'] = df['city'].ffill()
# backward fill: наступним значенням
df['city'] = df['city'].bfill()

# Заповнення по групах (середнє GPA по місту)
df['gpa'] = df.groupby('city')['gpa'].transform(
    lambda x: x.fillna(x.mean())
)
```

**Дублікати, типи, перетворення**

```
# ── Дублікати ──
print(df.duplicated().sum())          # кількість дублікатів
print(df[df.duplicated(keep=False)])  # показати всі дублюючі рядки
df = df.drop_duplicates()             # видалити дублікати
df = df.drop_duplicates(subset=['name', 'city'])  # за конкретними стовпцями

# ── Зміна типів даних ──
df['age'] = df['age'].astype(int)
df['gpa'] = df['gpa'].astype(float).round(2)
df['date'] = pd.to_datetime(df['date_str'], format='%d.%m.%Y')
df['category'] = df['city'].astype('category')  # економія пам'яті

# pd.to_numeric – безпечна конвертація
df['score'] = pd.to_numeric(df['score_str'], errors='coerce')
# errors='coerce': нечислові значення → NaN (не падає з помилкою)

# ── Перейменування стовпців ──
df = df.rename(columns={'name': 'full_name', 'gpa': 'grade_avg'})
df.columns = [c.lower().replace(' ', '_') for c in df.columns]

# ── Додавання та видалення стовпців ──
df['gpa_category'] = pd.cut(df['gpa'],
    bins=[0, 2.5, 3.5, 4.5, 5.0],
    labels=['F', 'C', 'B', 'A']
)
df['is_honors'] = df['gpa'] >= 4.5
df['name_upper'] = df['full_name'].str.upper()
df = df.drop(columns=['name_upper'])  # видалити стовпець

# ── Сортування ──
```

```python
df = df.sort_values('gpa', ascending=False)
df = df.sort_values(['year', 'gpa'], ascending=[True, False])
df = df.reset_index(drop=True)   # скинути індекс після сортування
```

## Apply, map та vectorized-операції

Apply дозволяє застосовувати довільну функцію до кожного елемента, рядка або стовпця DataFrame.

```python
import pandas as pd

df = pd.DataFrame({
    'name':  ['Іван Петренко', 'Марія Коваль', 'Олег Шевченко'],
    'score': [85, 92, 67],
    'city':  ['Київ', 'Харків', 'Київ'],
})

# ── .map() – перетворення значень у Series ──
city_code = {'Київ': 'KV', 'Харків': 'KH', 'Одеса': 'OD'}
df['city_code'] = df['city'].map(city_code)

# ── .apply() до стовпця (axis=0 за замовч.) ──
def score_to_grade(score):
    if score >= 90: return 'A'
    if score >= 75: return 'B'
    if score >= 60: return 'C'
    return 'F'

df['grade'] = df['score'].apply(score_to_grade)
# або через lambda:
df['grade'] = df['score'].apply(lambda x: 'A' if x >= 90 else 'B')

# ── .apply() до рядків (axis=1) ──
def full_info(row):
    return f"{row['name']} ({row['city']}): {row['grade']}"

df['info'] = df.apply(full_info, axis=1)

# ── Vectorized str-методи ──
df['first_name'] = df['name'].str.split().str[0]
df['last_name']  = df['name'].str.split().str[-1]
df['name_len']   = df['name'].str.len()
df['name_upper'] = df['name'].str.upper()
df['has_pet']    = df['name'].str.contains('Олег')

# ── numpy.where: умовна логіка (швидше ніж apply) ──
import numpy as np
df['level'] = np.where(df['score'] >= 80, 'strong', 'weak')

# numpy.select: кілька умов
conditions = [df['score'] >= 90, df['score'] >= 75, df['score'] >= 60]
choices    = ['A', 'B', 'C']
df['letter'] = np.select(conditions, choices, default='F')
```

## Групування та агрегація (groupby)

groupby – один з найпотужніших інструментів Pandas. Він розбиває DataFrame на групи за значенням стовпця і застосовує агрегуючу функцію до кожної групи.

```python
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'dept':    ['CS','CS','Math','Math','CS','Math','Phys'],
    'year':    [1, 2, 1, 3, 2, 2, 1],
    'gpa':     [4.2, 4.8, 3.5, 4.6, 3.9, 4.1, 3.7],
    'credits': [30, 60, 30, 90, 60, 60, 30],
})

# ── Базові агрегати ──
print(df.groupby('dept')['gpa'].mean())
# CS      4.300
# Math    4.067
# Phys    3.700

print(df.groupby('dept')['gpa'].agg(['mean', 'std', 'min', 'max', 'count']))

# ── Групування по кількох стовпцях ──
result = df.groupby(['dept', 'year'])['gpa'].mean()
print(result)

# ── .agg() - різні функції для різних стовпців ──
stats = df.groupby('dept').agg(
    avg_gpa    = ('gpa', 'mean'),
    max_gpa    = ('gpa', 'max'),
    total_cred = ('credits', 'sum'),
    students   = ('gpa', 'count'),
)
print(stats)

# ── .transform() - повертає Series тієї ж довжини ──
# Додаємо середній GPA кафедри для кожного студента
df['dept_avg_gpa'] = df.groupby('dept')['gpa'].transform('mean')
df['above_dept_avg'] = df['gpa'] > df['dept_avg_gpa']

# ── .filter() - фільтрація груп ──
# Залишити тільки кафедри з середнім GPA > 4.0
df_filtered = df.groupby('dept').filter(lambda g: g['gpa'].mean() > 4.0)

# ── pivot_table - зведена таблиця (як в Excel) ──
pivot = pd.pivot_table(
    df,
    values='gpa',
    index='dept',
    columns='year',
    aggfunc='mean',
    fill_value=0
)
print(pivot)

# ── value_counts() ──
print(df['dept'].value_counts())
print(df['year'].value_counts(normalize=True))  # відсотки
```

## Злиття та об'єднання таблиць

`merge()` – аналог SQL JOIN: об'єднує два DataFrame за спільним стовпцем.
`concat()` – «склеює» DataFrames вертикально або горизонтально.

| Метод | SQL-аналог / Дія |
|---|---|
| `merge(how='inner')` | SQL INNER JOIN: тільки рядки з обох таблиць де є збіг. |
| `merge(how='left')` | SQL LEFT JOIN: всі рядки лівої + збіги правої. |
| `merge(how='right')` | SQL RIGHT JOIN: збіги лівої + всі рядки правої. |
| `merge(how='outer')` | SQL FULL OUTER JOIN: всі рядки обох таблиць. |
| `concat(axis=0)` | Вертикальне склеювання (додати рядки). |
| `concat(axis=1)` | Горизонтальне склеювання (додати стовпці). |
| `join()` | Спрощена версія merge() по індексу. |

```python
import pandas as pd

students = pd.DataFrame({
    'student_id': [1, 2, 3, 4, 5],
    'name':       ['Іван','Марія','Олег','Ліна','Тарас'],
    'dept_id':    [1, 1, 2, 2, None],
})

departments = pd.DataFrame({
    'dept_id': [1, 2, 3],
    'dept':    ['CS', 'Math', 'Physics'],
})

grades = pd.DataFrame({
    'student_id': [1, 1, 2, 3, 4],
    'subject':    ['Math','CS','Math','CS','Physics'],
    'grade':      [90, 85, 95, 78, 88],
})

# ── INNER JOIN ──
merged = pd.merge(students, departments,
    on='dept_id', how='inner')
print(merged.shape)   # (4, 4)  - студент без dept_id виключений

# ── LEFT JOIN ──
merged_left = pd.merge(students, departments,
    on='dept_id', how='left')
# Тарас залишається, dept = NaN

# ── Різні назви ключових стовпців ──
# pd.merge(df1, df2, left_on='sid', right_on='student_id')

# ── Три таблиці ──
full = (
    students
    .merge(departments, on='dept_id', how='left')
    .merge(grades, on='student_id', how='left')
)
print(full.head())

# ── concat: об'єднати кілька CSV ──
import glob
files = glob.glob('data/monthly_*.csv')
dfs = [pd.read_csv(f) for f in files]
```

```
all_data = pd.concat(dfs, ignore_index=True)

# ── concat: горизонтально ──
df_wide = pd.concat([students, grades[['grade']]], axis=1)
```

### Робота з датами та часовими рядами

```
import pandas as pd

# ── Парсинг дат ──
df = pd.DataFrame({
    'date_str': ['01.09.2023', '15.10.2023', '20.11.2023', '05.01.2024'],
    'sales':    [1500, 2200, 1800, 3100],
})

df['date'] = pd.to_datetime(df['date_str'], format='%d.%m.%Y')

# ── .dt accessor – атрибути дати ──
df['year']      = df['date'].dt.year
df['month']     = df['date'].dt.month
df['day']       = df['date'].dt.day
df['weekday']   = df['date'].dt.day_name()  # 'Monday', ...
df['quarter']   = df['date'].dt.quarter
df['week']      = df['date'].dt.isocalendar().week

# ── Фільтрація за датами ──
after_oct = df[df['date'] >= '2023-10-01']
q4 = df[(df['date'] >= '2023-10-01') & (df['date'] < '2024-01-01')]

# ── Групування по місяцю ──
monthly = df.groupby(df['date'].dt.to_period('M'))['sales'].sum()
print(monthly)

# ── Resample: перегрупування часових рядів ──
df = df.set_index('date')
quarterly_sales = df['sales'].resample('Q').sum()
monthly_avg     = df['sales'].resample('ME').mean()

# ── Різниця між датами ──
df['days_since'] = (pd.Timestamp.now() - df.index).days
```

### Статистика та базова візуалізація

```
import pandas as pd
import matplotlib.pyplot as plt

# Генерація тестових даних
import numpy as np
np.random.seed(42)
df = pd.DataFrame({
    'age':    np.random.randint(18, 35, 100),
    'gpa':    np.random.uniform(2.5, 5.0, 100).round(2),
    'dept':   np.random.choice(['CS','Math','Physics'], 100),
    'year':   np.random.randint(1, 5, 100),
})

# ── Кореляція ──
print(df[['age', 'gpa', 'year']].corr())
```

```python
print(df['age'].corr(df['gpa']))   # кореляція двох стовпців

# ── Вбудовані графіки Pandas ──

# Гістограма
df['gpa'].hist(bins=20, figsize=(7, 4), color='steelblue', edgecolor='white')
plt.title('Розподіл GPA студентів')
plt.xlabel('GPA'); plt.ylabel('Кількість')
plt.tight_layout(); plt.savefig('gpa_hist.png', dpi=150); plt.show()

# Boxplot по групах
df.boxplot(column='gpa', by='dept', figsize=(8, 5))
plt.title('GPA за кафедрами'); plt.suptitle('')
plt.tight_layout(); plt.savefig('gpa_boxplot.png', dpi=150); plt.show()

# Bar chart
dept_avg = df.groupby('dept')['gpa'].mean().sort_values(ascending=False)
dept_avg.plot(kind='bar', figsize=(7, 4), color=['#2196F3','#4CAF50','#FF9800'])
plt.title('Середній GPA по кафедрах')
plt.xticks(rotation=0); plt.tight_layout()
plt.savefig('dept_bar.png', dpi=150); plt.show()

# Scatter plot: age vs gpa
df.plot(kind='scatter', x='age', y='gpa', alpha=0.5,
        c='year', colormap='viridis', figsize=(7, 5))
plt.title('Вік vs GPA (колір – курс)')
plt.tight_layout(); plt.savefig('scatter.png', dpi=150); plt.show()

# ── Кілька графіків на одному полотні ──
fig, axes = plt.subplots(2, 2, figsize=(10, 8))
fig.suptitle('Аналіз успішності студентів', fontsize=14)

df['gpa'].hist(ax=axes[0,0], bins=15, color='steelblue')
axes[0,0].set_title('Розподіл GPA')

dept_avg.plot(kind='bar', ax=axes[0,1], color='coral')
axes[0,1].set_title('Сер. GPA по кафедрах'); axes[0,1].tick_params(rotation=0)

df.plot(kind='scatter', x='age', y='gpa', ax=axes[1,0], alpha=0.4)
axes[1,0].set_title('Вік vs GPA')

df['year'].value_counts().sort_index().plot(kind='bar', ax=axes[1,1], color='green')
axes[1,1].set_title('Студентів по курсах')

plt.tight_layout(); plt.savefig('dashboard.png', dpi=150); plt.show()
```

## Завдання на лабораторну роботу

Встановіть бібліотеки:

```
pip install pandas openpyxl matplotlib seaborn
```

Для кожного завдання підготуйте тестові дані (або скористайтеся готовими CSV-файлами). Всі графіки зберігайте у папці lab15_output/ у форматах PNG та PDF.

15.1.Читання, перегляд та базова статистика.
Створіть CSV-файл students.csv із мінімум 30 рядками: name, age, year, dept, gpa, city, enrolled_date. Деякі значення навмисно залиште порожніми (NaN). Виконайте:

- Прочитайте файл через pd.read_csv(), виведіть: shape, dtypes, head(5), tail(5), info(), describe().
- Виведіть кількість та відсоток пропусків у кожному стовпці.
- Заповніть пропуски: gpa – медіаною, city – модою ('Невідомо' якщо всі різні), age – середнім.
- Виведіть студентів старше 21 року з GPA >= 4.0, відсортованих за GPA за спаданням.
- Побудуйте гістограму розподілу GPA та boxplot GPA по роках навчання.

15.2. Фільтрація, трансформація та вибірка.
Використайте той самий датасет students.csv або створіть новий. Виконайте:

- Додайте стовпець gpa_letter через np.select: A (>=4.5), B (>=3.5), C (>=2.5), F (<2.5).
- Додайте стовпець enrolled_year через .dt.year після парсингу enrolled_date.
- Через .str-методи: виведіть студентів, чиє ім'я починається на літеру 'М'; додайте стовпець first_name (перше слово імені).
- Знайдіть та видаліть дублікати за комбінацією name + enrolled_date.
- Використайте query() для знаходження студентів 2 або 3 курсу з Києва або Харкова.
- Відсортуйте за (year, gpa DESC) і збережіть результат у filtered_students.csv.

15.3. Групування та зведені таблиці. Виконайте групування та агрегацію:

- По кафедрі: count, mean GPA, std GPA, max GPA – через .agg().
- По року навчання: середній вік та середній GPA.
- Побудуйте pivot_table: рядки – кафедра, стовпці – рік навчання, значення – середній GPA.
- Знайдіть кафедру з найвищим середнім GPA.
- Додайте стовпець dept_rank – ранг студента всередині своєї кафедри за GPA (1 = найкращий): df.groupby('dept')['gpa'].rank(ascending=False).
- Побудуйте grouped bar chart: для кожної кафедри – стовпці по роках навчання.

15.4. Злиття таблиць та часові ряди. Створіть три CSV-файли:

- students.csv: student_id, name, dept_id, city.
- departments.csv: dept_id, dept_name, head_name, building.
- grades.csv: student_id, subject, grade, exam_date.

Виконайте:

- LEFT JOIN students + departments; перевірте студентів без кафедри (NaN після merge).
- Об'єднайте з grades через INNER JOIN; студенти без оцінок не потрапляють.
- Порахуйте середню оцінку кожного студента та додайте до основного DataFrame.
- Проаналізуйте динаміку оцінок по місяцях: groupby(exam_date.dt.to_period('M')).
- Знайдіть студентів з покращенням оцінок у другому семестрі відносно першого.
- Збережіть фінальну зведену таблицю в Excel з двома аркушами: 'Студенти' та 'Статистика по кафедрах'.

## Контрольні запитання

1. Що таке аналіз даних? Поясніть шість етапів циклу аналізу даних своїми словами. Чому очищення даних займає до 80% часу?
2. Чим DataFrame відрізняється від Series? Яке відношення між ними? Як звернутися до одного стовпця DataFrame двома різними способами?
3. Яка різниця між .loc[] та .iloc[]? Що поверне df.loc[2] і df.iloc[2] якщо індекс DataFrame починається з 10?
4. Що таке NaN у Pandas? Назвіть три способи обробки пропусків. Коли краще видаляти рядки з NaN, а коли заповнювати?
5. Що таке векторизована операція? Чому df['gpa'] * 20 є кращим підходом ніж цикл for? Яку роль відіграє NumPy?
6. Поясніть роботу groupby(). Що відбувається коли ви викликаєте df.groupby('dept')['gpa'].mean()? Чим .transform() відрізняється від .agg()?
7. Назвіть чотири типи злиття (merge) та поясніть кожен на прикладі. Чим merge() відрізняється від concat()?
8. Що таке кореляція? Як інтерпретувати значення коефіцієнта кореляції 0.9, 0.1 та -0.8? Чому кореляція не означає причинно-наслідковий зв'язок?
9. Що таке .dt accessor? Назвіть три атрибути дати, доступні через нього. Як згрупувати дані по місяцях?
10. Що таке EDA (Exploratory Data Analysis)? Які типи графіків найкраще підходять для: розподілу числової змінної, порівняння категорій, виявлення кореляції між двома числовими змінними?