

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
МИКОЛАЇВСЬКИЙ НАЦІОНАЛЬНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ МЕНЕДЖМЕНТУ

Кафедра економічної кібернетики,
комп'ютерних наук та інформаційних технологій

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Конспект лекцій

для здобувачів першого (бакалаврського) рівня вищої освіти
ОПП «Комп'ютерні науки» спеціальності F3 (122) «Комп'ютерні науки»
денної форми здобуття вищої освіти



УДК 004.42
А45

Друкується за рішенням науково-методичної комісії факультету менеджменту Миколаївського національного аграрного університету (протокол №1 від 28 серпня 2025 року)

Укладачі:

О. Ю. Пархоменко – к.ф.-м.н., доцент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

С. І. Тищенко – в.о. завідувача кафедри, к.п.н., доцент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

С. І. Ємельянов – PhD, старший викладач кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

О. О. Жебко – асистент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

О. Є. Богатенкова – асистент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету.

Рецензенти:

В.В.Базаренко – заступник начальника Миколаївської обласної військової адміністрації з питань цифрового розвитку, цифрових трансформацій і цифровізації (CDTO);

Д.Л.Кошкін – к.т.н., дцент, доцент кафедри кафедри електроенергетики, електротехніки та електромеханіки Миколаївського національного аграрного університету.

Алгоритмізація та програмування : конспект лекцій для здобувачів
А45 першого (бакалаврського) рівня вищої освіти ОПП «Комп'ютерні науки» спеціальності F3 (122) «Комп'ютерні науки» денної форми здобуття вищої освіти / уклад. О. Ю. Пархоменко, С. І. Тищенко, С. І. Ємельянов, О. О. Жебко, О. Є. Богатенкова. Миколаїв : МНАУ, 2025. 550 с.

УДК 004.42

© Миколаївський національний
аграрний університет, 2025

ЗМІСТ

ПЕРЕДМОВА.....	4
ЛЕКЦІЯ 1 Вступ до програмування і мови Python. Змінні та типи даних.....	6
ЛЕКЦІЯ 2 Базові оператори та введення/виведення даних.....	16
ЛЕКЦІЯ 3 Умовні конструкції та розгалуження програми	31
ЛЕКЦІЯ 4 Цикли: for та while	48
ЛЕКЦІЯ 5 Вкладені цикли та складні алгоритми	67
ЛЕКЦІЯ 6 Робота з рядками.....	89
ЛЕКЦІЯ 7 Списки та кортежі.....	112
ЛЕКЦІЯ 8 Словники та множини	142
ЛЕКЦІЯ 9 Основи функцій у Python.....	178
ЛЕКЦІЯ 10 Розширені можливості функцій	204
ЛЕКЦІЯ 11 Модулі, пакети та робота з файлами.....	233
ЛЕКЦІЯ 12 Базові структури даних та алгоритми: стек і черга, сортування, пошук	269
ЛЕКЦІЯ 13 Використання колекцій Python для роботи з даними	304
ЛЕКЦІЯ 14 Класи та об'єкти. Основи ООП	341
ЛЕКЦІЯ 15 Наслідування та поліморфізм.....	373
ЛЕКЦІЯ 16 Інкапсуляція та спеціальні методи класів.....	411
ЛЕКЦІЯ 17 Графічна бібліотека Turtle та візуалізація даних з Matplotlib....	456
ЛЕКЦІЯ 18 Створення GUI з Tkinter та CustomTkinter	500
ПІСЛЯМОВА	546
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	548

ПЕРЕДМОВА

Шановні студенти! Ви тримаєте в руках (або переглядаєте на екрані) конспект лекцій, який стане вашим надійним провідником у захопливий світ програмування на Python. Цей курс створено спеціально для студентів спеціальності "Комп'ютерні науки" з урахуванням сучасних вимог до підготовки ІТ-фахівців та тенденцій розвитку індустрії програмного забезпечення.

Мова програмування Python сьогодні є однією з найпопулярніших у світі. Вона використовується у веб-розробці, науці про дані, машинному навчанні, автоматизації, створенні ігор та багатьох інших сферах. За даними рейтингів ТЮВЕ та IEEE Spectrum, Python стабільно входить до трійки найпопулярніших мов програмування, а попит на фахівців, які володіють цією мовою, постійно зростає. Але головне – Python ідеально підходить для навчання програмуванню завдяки своєму чистому, читабельному синтаксису та величезній спільноті розробників.

Як побудований цей конспект лекцій?

Матеріал структуровано таким чином, щоб забезпечити поступове, глибоке та системне засвоєння знань. Ми рухаємося від простого до складного: від написання першої програми "Hello, World!" до створення повноцінних графічних додатків. Кожна лекція містить не лише теоретичний матеріал, але й велику кількість практичних прикладів коду, які ілюструють розглянуті концепції.

Особливістю цього посібника є акцент на розумінні фундаментальних принципів, а не на механічному запам'ятовуванні синтаксису. Ми детально розглядаємо не лише те, "як" це працює, але й "чому" це працює саме так. Саме тому після кожної лекції ви знайдете питання для обговорення, які допоможуть закріпити матеріал, розвинути критичне мислення та навчитися застосовувати знання в нестандартних ситуаціях.

Що ви вивчите?

Курс охоплює всі ключові теми, необхідні для початку професійної кар'єри:

- **Основи програмування:** змінні, типи даних, оператори, умовні конструкції, цикли – все те, що формує фундамент алгоритмічного мислення.

- **Структури даних:** рядки, списки, кортежі, словники, множини. Ви навчитеся обирати правильну структуру даних для розв'язання конкретної задачі.

- **Функції та модульність:** створення власних функцій, використання стандартної бібліотеки, організація коду в модулі та пакети.

- **Алгоритми та структури даних:** стеки, черги, сортування, пошук. Ви не просто навчитеся використовувати вбудовані функції, а й зрозумієте, як вони працюють "під капотом".

- **Об'єктно-орієнтоване програмування:** класи, об'єкти, інкапсуляція, наслідування, поліморфізм – парадигма, яка лежить в основі більшості сучасних програмних систем.

- **Графічні інтерфейси:** створення віконних додатків за допомогою Tkinter та сучасного CustomTkinter.

- **Візуалізація даних:** побудова графіків та діаграм за допомогою бібліотеки Matplotlib.

Як працювати з конспектом лекцій?

Найкращий спосіб опанувати програмування – це **практика**. Недостатньо просто прочитати текст – обов'язково запускате всі приклади, експериментуйте з кодом, змінюйте його, ламайте і виправляйте. Спробуйте відповісти на питання для обговорення не лише подумки, але й записуючи свої думки та пишучи невеликі програми для перевірки гіпотез.

Пам'ятайте, що програмування – це не лише знання синтаксису, але й спосіб мислення. Вміння розбивати складну задачу на прості, знаходити закономірності, будувати алгоритми – це навички, які розвиваються лише з практикою. Не бійтеся помилок – вони є невід'ємною частиною навчального процесу. Кожна помилка, яку ви виправите, робить вас кращим програмістом.

Сподіваюся, що цей посібник стане для вас не просто підручником, а настільною книгою, до якої ви будете повертатися навіть після завершення курсу. Бажаю вам успіхів, натхнення та задоволення від процесу створення програм! Пам'ятайте: кожен експерт колись був початківцем, який не зупинився на половині шляху.

Тож відкривайте першу лекцію, запускайте Python і почнемо цю захопливу подорож!

ЛЕКЦІЯ 1

Вступ до програмування і мови Python.

Змінні та типи даних

План лекції

1. Вступ до світу програмування: що таке алгоритм та програма?

Тут ми закладаємо фундаментальні поняття, які є універсальними для будь-якої мови, пояснюємо, як комп'ютер "розуміє" команди, та знайомимо з концепцією мов програмування як посередника.

2. Мова Python: історія, філософія, "за" і "проти" та магія першої програми.

Розповідаємо про те, чому саме Python став таким популярним, які його сильні та слабкі сторони, і одразу занурюємо студента в практику – пишемо "Hello, World!", пояснюючи кожен символ.

3. Ваш перший інструментарій: встановлення Python, налаштування середовища та знайомство з інтерпретатором.

Детальний покроковий гайд з встановлення на різні ОС, огляд IDE та редакторів коду, пояснення різниці між інтерпретатором та компілятором, щоб студент одразу міг почати писати код самостійно.

4. Будова програми: змінні як комірки пам'яті, базові типи даних та правила гарного тону (PEP 8).

Переходимо до основ синтаксису: як створюються змінні, чому вони потрібні, які бувають типи даних, як правильно давати назви, щоб код був чистим і зрозумілим.

1. Вступ до світу програмування: що таке алгоритм та програма?

Перш ніж ми почнемо писати код мовою Python, важливо зрозуміти фундаментальні концепції, які лежать в основі будь-якої мови програмування. Адже Python – це лише інструмент, а наша мета – навчитися за допомогою цього інструменту вирішувати задачі. Уявіть, що ви кухар. Мова програмування – це ваш набір кухонного начиння. Але щоб приготувати страву (програму), потрібен рецепт. У світі програмування цим рецептом є **алгоритм**.

Що таке алгоритм?

Алгоритм – це точний, скінченний набір інструкцій, який описує послідовність дій виконавця для досягнення певної мети або розв'язання задачі.

Це поняття набагато старше за комп'ютери. Найпростіший приклад алгоритму з нашого повсякденного життя – це рецепт приготування кави:

1. Взяти чашку.
2. Насипати 1-2 чайні ложки кави.
3. Додати цукор за смаком.
4. Залити окропом.
5. Перемішати.
6. Якщо хочете з молоком, додати молоко.

Цей опис задовольняє всім вимогам до алгоритму:

- **точність та дискретність:** кожен крок (інструкція) чіткий і зрозумілий.
- **скінченність:** алгоритм обов'язково завершиться після виконання всіх кроків.

- **результативність:** ми отримаємо готову каву.
- **масовість:** за цим рецептом можна приготувати каву багато разів.

Так само і в програмуванні: ми створюємо алгоритми для обчислення складних математичних формул, пошуку потрібної інформації в Інтернеті, керування рухом робота або обробки фотографій.

Від алгоритму до програми

Комп'ютер – це ідеальний виконавець, який блискавично виконує інструкції, але він не розуміє людської мови. Процесор комп'ютера оперує лише машинним кодом – послідовностями нулів та одиниць (двійковий код). Наприклад, інструкція для додавання двох чисел у машинному коді може виглядати як 10110000 01100001. Людині писати такі інструкції вкрай складно та неефективно.

Саме тут на допомогу приходять **мови програмування**. Вони є проміжною ланкою між людиною та машиною. Ми пишемо інструкції зрозумілою для нас мовою (наприклад, `result = 2 + 2`), а спеціальна програма перетворює це на зрозумілий комп'ютеру машинний код.

Таким чином, **програма** – це реалізація алгоритму засобами конкретної мови програмування. Це текст (код), який складається з команд, що згодом будуть виконані комп'ютером. Програма може складатися з десятків або мільйонів рядків коду, але в її основі завжди лежить чіткий алгоритм.

Розуміння цієї різниці є ключовим для початківця. Алгоритм – це **що** треба зробити (стратегія), а програма – це **як саме** ми пояснимо це комп'ютеру (тактика). Вивчаючи Python, ми зосередимось на тактиці, але постійно будемо тримати в голові стратегію – алгоритм, який реалізуємо.

2. Мова Python: історія, філософія, "за" і "проти" та магія першої програми

Чому з-поміж сотень мов програмування ми обрали саме Python? Відповідь криється в його історії та філософії, які зробили його одним із найпопулярніших інструментів у світі.

Історія створення

Наприкінці 1980-х років нідерландський програміст Гвідо ван Россум працював над мовою програмування ABC. Він бачив її сильні сторони (легкість навчання, зрозумілий синтаксис), але також помічав і недоліки. Йому хотілося створити мову, яка була б такою ж зручною, як ABC, але більш потужною та універсальною. У 1989 році, під час різдвяних канікул, він почав роботу над нащадком. Назву для проекту він обрав не на честь змії, а як данину поваги улюбленому британському комедійному серіалу "Монті Пайтон і Священний Грааль". Так з'явився Python.

Філософія та "за"

Гвідо ван Россум заклав у Python декілька ключових принципів, які пізніше оформилися як "Дзен Python" (PEP 20). Їх можна прочитати, ввівши в інтерпретаторі `import this`. Найголовніші з них:

- **Красиве краще ніж потворне.**
- **Явне краще ніж неявне.**
- **Просте краще ніж складне.**
- Це втілюється в надзвичайно **чистий та читабельний синтаксис**. Блоки коду визначаються відступами, а не фігурними дужками, як у C++ або Java. Це змушує програмістів писати акуратно структурований код.

Основні переваги Python:

1. **Простота вивчення:** Python дуже схожий на звичайну англійську мову. Це робить його ідеальним вибором для початківців.

2. **Величезна спільнота та екосистема:** Навколо Python сформувалося потужне ком'юніті. Це означає, що на будь-яке питання ви знайдете відповідь на Stack Overflow. Але головне – це тисячі бібліотек (готових наборів інструментів) на будь-який смак: для веб-розробки (Django, Flask), для науки про дані (Pandas, NumPy), для машинного навчання (TensorFlow, PyTorch), для автоматизації та навіть для створення ігор (Pygame).

3. **Універсальність:** Python використовується всюди: від написання простих скриптів для автоматизації нудної роботи до створення бекенду

гігантських веб-сайтів (YouTube, Instagram, Spotify) і найскладніших систем штучного інтелекту.

Недоліки Python

Звісно, ідеальних мов не існує.

1. **Швидкість виконання:** Python – це інтерпретована мова, і він працює повільніше, ніж компільовані мови, такі як C++ або Java. Для більшості задач, з якими ви зіткнетесь на початку навчання, ця різниця буде непомітною. У високонавантажених системах "повільні" частини коду можна писати на C++ і підключати до Python.

2. **Споживання пам'яті:** Python може споживати більше оперативної пам'яті, ніж деякі інші мови, через особливості внутрішньої організації об'єктів.

Магія першої програми

Не зважаючи на ці недоліки, переваги Python роблять його беззаперечним лідером для навчання та швидкої розробки. Давайте відчуємо цю простоту, написавши нашу першу програму. Вона буде виводити на екран класичне вітання "Hello, World!".

Відкрийте ваш текстовий редактор (про його вибір та налаштування ми поговоримо в наступному пункті) та наберіть один єдиний рядок:

```
print("Hello, World!")
```

Запустіть програму. Ви побачите:

Hello, World!

Вітання! Ви щойно написали свою першу програму. Розберемо, що відбулося:

- `print` – це **вбудована функція** Python. Функція – це маленька програма всередині Python, яка вміє робити щось корисне. У цьому випадку функція `print` вміє виводити текст на екран.
- Дужки `()` – в них ми передаємо функції те, що хочемо надрукувати. Це називається **аргументом** або параметром функції.
- `"Hello, World!"` – це **рядок** (тип даних `str`, `string`). Рядки завжди беруться в лапки, щоб Python розумів, де текст, а де команди.

Цей простий приклад ілюструє головну ідею Python: команди виглядають майже як звичайні речення. У наступних лекціях ми будемо ускладнювати ці речення, додаючи до них змінні, цикли та умови, створюючи справді потужні програми. Ви зробили перший крок – найважливіший і найскладніший. Попереду багато цікавого!

3. Ваш перший інструментарій: встановлення Python, налаштування середовища та знайомство з інтерпретатором

Теорія – це чудово, але програмування – це насамперед практика. Щоб писати код, нам потрібно підготувати "робоче місце". Цей процес називається налаштуванням середовища розробки. Не лякайтеся, це набагато простіше, ніж здається. Ми встановимо сам Python та оберемо зручний інструмент для написання коду – інтегроване середовище розробки (IDE) або текстовий редактор.

Встановлення Python

Перш за все, сам Python потрібно "поселити" на ваш комп'ютер. На відміну від більшості програм, Python не встановлюється за замовчуванням у Windows (на macOS та Linux він часто вже є, але краще встановити свіжу версію).

1. **Завантаження:** Відкриваємо браузер і переходимо на офіційний сайт python.org. Наводимо курсор на розділ "Downloads". Сайт розумний – він сам визначить вашу операційну систему і запропонує потрібну кнопку. Сміливо натискаємо жовту кнопку "Download Python X.XX" (де X.XX – номер останньої стабільної версії, наприклад, 3.12).

2. Встановлення на Windows:

- Запускаємо завантажений .exe файл.
- **ВАЖЛИВО!** У самому низу вікна встановлювача **обов'язково поставте галочку** "Add Python X.XX to PATH". Це дозволить вам запускати Python з будь-якої папки в командному рядку.
- Виберіть "Install Now" (рекомендовано) або "Customize installation", якщо хочете змінити шлях встановлення. Зачекайте кілька хвилин.
- Після завершення ви побачите повідомлення "Setup was successful". Для перевірки натисніть Win + R, введіть cmd і в командному рядку наберіть `python --version`. Ви маєте побачити щойно встановлену версію.

3. **Встановлення на macOS та Linux:** Процес трохи відрізняється, але на сайті python.org є детальні інструкції. Найчастіше на macOS використовують встановлювач .pkg, а в Linux – менеджер пакетів (наприклад, `sudo apt install python3` для Ubuntu/Debian).

Огляд середовищ розробки (IDE та редактори)

Коли Python встановлено, нам потрібне місце, де ми будемо писати код. Можна писати в звичайному "Блокноті", але це дуже незручно. Сучасні

інструменти пропонують підсвітку синтаксису, підказки, автоматичне форматування та багато інших "плюшок". Є два основні типи інструментів: IDE та редактори коду.

- **IDE (Integrated Development Environment)** – це "важкі" багатофункціональні комбайни, які містять усе необхідне для розробки "з коробки". Найпопулярніший для Python – це **PyCharm** від компанії JetBrains. Він буває у безкоштовній версії (Community Edition) та платній (Professional). PyCharm розуміє структуру вашого проекту, має вбудований налагоджувач (debugger), систему контролю версій і багато іншого. Для початківця він може здатися трохи громіздким, але це дуже потужний інструмент.

- **Редактори коду** – це легші, але швидкі та гнучкі інструменти. Беззаперечний лідер тут – **Visual Studio Code (VS Code)** від Microsoft. Він безкоштовний, працює на всіх платформах, і його функціональність розширюється за допомогою плагінів (розширень). Для роботи з Python вам знадобиться встановити розширення "Python" від Microsoft. VS Code ідеально підходить для навчання: він швидкий, зрозумілий, але при цьому надзвичайно потужний.

- **Альтернативи:**

- **IDLE:** це найпростіше середовище, яке встановлюється разом з Python. Воно дуже базове, але для написання перших 5-10 рядків коду цілком підходить.

- **Jupyter Notebook:** це веб-орієнтоване інтерактивне середовище, яке дозволяє писати код невеликими блоками (комірками) і одразу бачити результат. Воно надзвичайно популярне в Data Science, але для вивчення основ синтаксису та створення великих програм підходить менше через свою специфічну структуру.

Поняття інтерпретатора

Коли ми пишемо код у VS Code або PyCharm, нам потрібен хтось, хто "прочитає" наш текст і перетворить його на дії комп'ютера. Цю роль виконує **інтерпретатор Python**. Це та сама програма, яку ми встановили з python.org.

Тут варто згадати різницю між **компілятором** та **інтерпретатором**:

- **Компілятор (C++, Java):** бере весь ваш код і перекладає його на машинну мову за один раз, створюючи окремий виконуваний файл (.exe). Потім цей файл можна запускати без компілятора. Це як перекладач, який спочатку повністю перекладає книгу, а потім ви її читаете.

- **Інтерпретатор (Python, JavaScript):** бере ваш код рядок за рядком, одразу перекладає його і виконує. Якщо в коді є помилка на 10-му рядку,

програма виконає перші 9 і зупиниться. Це як перекладач-синхроніст, який перекладає мову спікера в реальному часі.

Python є інтерпретованою мовою. Це дозволяє нам писати код і одразу бачити результат, що дуже зручно для навчання.

Запуск першої програми

Тепер об'єднаємо все разом. Відкрийте VS Code (або PyCharm). Створіть новий файл з розширенням .py, наприклад my_first_program.py. Наберіть там print("Hello, World!"). Натисніть кнопку "Run" (або відкрийте термінал і напишіть python my_first_program.py). У нижній частині екрана з'явиться ваш довгоочікуваний напис. Вітаю, ваше середовище налаштоване і працює!

4. Будова програми: змінні як комірки пам'яті, базові типи даних та правила гарного тону (PEP 8)

Тепер, коли ми вміємо запускати код, настав час навчитися зберігати в ньому дані. Уявіть, що програма – це наш робочий стіл. Нам потрібні коробочки, в які ми складатимемо різні речі (числа, текст), щоб потім їх використовувати. У програмуванні ці коробочки називаються **змінними**.

Змінні (Variables)

Змінна – це іменована область пам'яті, призначена для зберігання даних. Можна думати про це як про коробку з наклейкою. На наклейці написано ім'я змінної, а всередині коробки лежить якесь значення. Python – це мова з **динамічною типізацією**. Це означає, що нам не потрібно заздалегідь оголошувати, що лежатиме в коробці (число чи текст). Python сам це зрозуміє.

Щоб створити змінну, потрібно просто придумати їй ім'я та присвоїти значення за допомогою оператора = (оператор присвоєння).

```
student_name = "Олена" # Створили змінну student_name і поклали в неї
рядок "Олена"
student_age = 19        # Створили змінну student_age і поклали в неї число 19
is_studying = True      # Створили змінну is_studying і поклали в неї булеве
значення True
```

Після цього ми можемо використовувати ці змінні в програмі:

```
print(student_name) # Виведе: Олена
print(student_age + 5) # Виведе: 24
```

Правила іменування змінних

Імена змінним не можна давати як попало. Є суворі правила і гарні манери.

- **Правила (інакше буде помилка):**

- Ім'я може містити лише літери (a-z, A-Z), цифри (0-9) та знак підкреслення `_`.

- Ім'я не може починатися з цифри. `1user` – помилка, `user1` – добре.

- Ім'я не може збігатися з ключовими (зарезервованими) словами Python. Наприклад, не можна назвати змінну `if`, `while`, `for`, `import`, `True` тощо. (VS Code підсвітить їх іншим кольором).

- Python – **регістрозалежна** мова. Змінні `name`, `Name` та `NAME` – це три різні змінні.

- **Конвенції (PEP 8 – якщо порушити, то не буде помилки, але ваш код вважатимуть "некрасивим"):**

- Для імен змінних та функцій використовується стиль `snake_case` – всі слова маленькими літерами через підкреслення. `my_super_variable`, `student_first_name`.

- Імена мають бути змістовними. `a = 10` – погано, `students_count = 10` – добре.

Базові типи даних

Як ви вже помітили, дані в змінних бувають різними. "Олена" – це текст, `19` – це число, `True` – це істина. Це і є типи даних.

1. **int (цілі числа):** Всі цілі числа без дробової частини. `10`, `-5`, `100500`.

2. **float (числа з плаваючою комою):** Дійсні числа, дробові числа. Важливо, що в програмуванні дріб позначається **через крапку**, а не кому. `3.14`, `-0.001`, `2.0` (це теж `float`, хоча число ціле).

3. **str (рядки, strings):** Послідовність символів (текст). Завжди беруться в лапки – одинарні ('Привіт') або подвійні ("Hello"). Для Python немає різниці, головне – бути послідовним.

4. **bool (булеві значення):** Логічний тип даних. Може мати лише два значення: `True` (істина) або `False` (хибність). Використовується в умовах.

Якщо ви не впевнені, якого типу даних ваша змінна, ви завжди можете це перевірити за допомогою вбудованої функції `type()`:

```
price = 99.99
print(type(price)) # Виведе: <class 'float'>
```

Приведення типів

Іноді потрібно перетворити один тип даних на інший. Наприклад, число на текст, або текст, який складається з цифр, на число. Це називається **приведенням (casting)**.

Користувач завжди вводить текст. Якщо ми чекаємо число, його треба перетворити.

```
age_str = input("Скільки вам років? ") # Це буде рядок (str)
age_int = int(age_str)                  # Перетворюємо рядок на ціле число (int)
print("Наступного року вам буде:", age_int + 1)
```

Можна перетворити і навпаки

```
price = 100
message = "Ціна товару: " + str(price) # Потрібно перетворити число в
рядок, щоб склеїти
```

Розуміння змінних та типів даних – це основа, на якій будується все інше. Як тільки ви навчитеся впевнено оперувати цими поняттями, можна рухатися далі – до обчислень та логічних операцій.

Питання для обговорення

1. Поясніть різницю між алгоритмом та програмою. Чи може існувати програма без алгоритму? Чому мови програмування називають "посередником" між людиною та комп'ютером?

2. Чому Python вважають мовою з "чистим та читабельним синтаксисом"? Наведіть приклади, чим синтаксис Python відрізняється від синтаксису C++ або Java (наприклад, у визначенні блоків коду).

3. Які основні переваги та недоліки Python порівняно з компільованими мовами програмування? У яких сферах Python є беззаперечним лідером, а в яких його використання може бути проблематичним?

4. Що таке інтерпретатор? Яка його роль у виконанні Python-програми? Поясніть різницю між інтерпретацією та компіляцією.

5. Чому важливо додавати Python до PATH під час встановлення? Що станеться, якщо цього не зробити, і як це можна виправити?

6. Яке середовище розробки (IDE або редактор) ви оберете для вивчення Python і чому? Які функції мають бути в ідеальному інструменті для початківця?

7. Проаналізуйте рядок коду: `student_age = 20`. Що таке `student_age`? Що таке 20? Що таке `=` у цьому контексті? Як би ви назвали цей рядок одним терміном?

8. Чому важливо давати змінним змістовні імена та дотримуватися стилю `snake_case`? Наведіть приклад "поганого" імені змінної та запропонуйте для нього "хорошу" альтернативу.

9. Уявіть, що виконується код: `result = 10 / 3`. Якого типу даних буде змінна `result`? Чому результат ділення цілих чисел може стати дробовим числом, і як це пов'язано з типами даних?

10. Чому виникає помилка: `print("Мені " + 5 + " років")`? Як її правильно виправити, використовуючи приведення типів? Напишіть правильний варіант.

ЛЕКЦІЯ 2

Базові оператори та введення/виведення даних

План лекції

1. Арифметичні оператори та пріоритети: Python як калькулятор.

Тут ми розглянемо всі математичні дії, які вміє виконувати Python: від звичайного додавання до піднесення до степеня та ділення з остачею. Особливу увагу приділимо пріоритету операцій – "шкільній" математиці, яка працює і в програмуванні.

2. Оператори порівняння та логічні оператори: як комп'ютер приймає рішення.

Навчимо програму порівнювати числа та рядки, використовувати логічні зв'язки "і", "або", "не" та будувати таблиці істинності. Це фундамент для майбутніх умовних конструкцій.

3. Присвоєння з операціями та спеціальні оператори (in, is).

Розберемо, як ефективно змінювати значення змінних (наприклад, `x += 5`), а також познайомимося з корисними операторами для перевірки наявності елемента в колекції (`in`) та порівняння тотожності об'єктів (`is`), пояснивши різницю від звичайного `==`.

4. Введення та виведення даних: спілкування з користувачем.

Практичний блок про те, як зробити програму інтерактивною: виводити дані красиво (зрозумімо `print()` глибше, з параметрами `sep` та `end`), отримувати дані від користувача через `input()` та навчимося формувати рядки сучасно та елегантно за допомогою `f-strings`.

1. Арифметичні оператори та пріоритети: Python як калькулятор

Комп'ютер зрештою був створений для швидких обчислень. Python успадкував цю здатність і пропонує нам набір інструментів – **арифметичних операторів** – для виконання математичних дій. Ви вже користувалися калькулятором, тепер навчимося користуватися Python як калькулятором, але набагато потужнішим.

У Python є сім базових арифметичних операторів. Даваймо розглянемо їх на прикладах. Уявімо, що ми готуємо програму для піцерії, де нам потрібно рахувати вартість замовлень, кількість інгредієнтів тощо.

`a = 15` *# кількість шматочків піци*

`b = 4` *# кількість людей*

1. Додавання (+): Найпростіший оператор. Додає два числа.

```
total_slices = a + 10 # докупили ще 10 шматків  
print(total_slices) # 25
```

До речі, + працює і з рядками (склеюючи їх), але це називається конкатенацією, про що ми поговоримо в лекції про рядки.

2. Віднімання (-): Віднімає одне число від іншого.

```
slices_eaten = 8  
slices_left = a - slices_eaten  
print(slices_left) # 7
```

3. Множення (*): Множить числа.

```
price_per_slice = 25  
total_cost = a * price_per_slice  
print(total_cost) # 375 (гривень)
```

4. Ділення (/): Результатом цього оператора завжди буде число з плаваючою комою (float).

```
slices_per_person = a / b  
print(slices_per_person) # 3.75 (кожному по 3.75 шматочки)
```

Зверніть увагу: навіть якщо числа діляться націло, ми отримуємо float (наприклад, 10 / 2 поверне 5.0, а не 5).

5. Цілочисельне ділення (//): Цей оператор повертає лише цілу частину від ділення, відкидаючи дробовий залишок. Дуже корисний, коли нам потрібна ціла кількість чогось.

```
whole_slices_per_person = a // b  
print(whole_slices_per_person) # 3 (кожен отримає по 3 цілих шматки, а  
решту ділитимуть)
```

6. Остача від ділення (%): Цей оператор часто називають "модуль" (від modulus). Він повертає не частку, а **залишок** від ділення. Це один з найкорисніших операторів, який широко використовується в алгоритмах (наприклад, для перевірки числа на парність).

```
leftover_slices = a % b  
print(leftover_slices) # 3 (15 поділити на 4 = 3 цілих і 3 в залишку)
```

Перевірка на парність

```
number = 10  
if number % 2 == 0:
```

```
print("Число парне") # Це буде виконано, бо 10 % 2 = 0
```

7. **Піднесення до степеня (**):** Зводить число в степінь.

Наприклад, ми хочемо порахувати кількість варіантів начинки для піци

```
toppings = 2
```

```
combinations = 2 ** toppings
```

```
print(combinations) # 4 (кожен інгредієнт може бути або не бути)
```

Пріоритет операцій

Коли у виразі зустрічається кілька операторів, Python виконує їх не просто зліва направо, а за чіткими правилами пріоритету (точно так само, як нас вчили в школі). Це важливо пам'ятати, щоб ваші обчислення були правильними.

1. **Дужки ()** мають найвищий пріоритет. Все, що в дужках, виконується в першу чергу.

2. **Піднесення до степеня **** (правоасоціативний, тобто $2 ** 2 ** 3$ виконається як $2 ** (2 ** 3)$).

3. **Унарний мінус** (наприклад, -5).

4. **Множення, ділення, цілочисельне ділення та остача: *, /, //, %** (вони мають однаковий пріоритет і виконуються зліва направо).

5. **Додавання та віднімання: +, -** (теж зліва направо).

Розглянемо приклад з нашого "піцерійного" життя:

Припустимо, у нас є 3 види піци по 100 грн і 2 види по 150 грн.

Ми хочемо порахувати загальну вартість, якщо беремо по одній кожного виду.

```
total_cost = 3 * 100 + 2 * 150 # Без дужок Python спочатку помножить  
(300 + 300 = 600)
```

```
print(total_cost) # 600
```

Але якщо ми хочемо спочатку додати кількість піц (3+2), а потім помножити на середню ціну?

Тоді дужки обов'язкові!

```
average_price = (3 + 2) * 125 # 5 * 125 = 625
```

```
print(average_price) # 625
```

Як бачите, Python – це не просто калькулятор, це дуже розумний калькулятор, який дотримується математичних правил. Розуміння цих операторів та їх пріоритетів – перший крок до написання обчислювальних програм.

2. Оператори порівняння та логічні оператори: як комп'ютер приймає рішення

Зберігати числа та виконувати обчислення – це добре, але справжня сила програмування проявляється тоді, коли програма може "думати" та приймати рішення на основі вхідних даних. Для цього нам потрібні оператори порівняння та логічні оператори. Результатом роботи цих операторів завжди є булеве значення: True (істина) або False (хибність).

Оператори порівняння

Вони використовуються для порівняння двох значень між собою. У Python їх вісім:

Оператор	Значення	Приклад (x=5, y=3)	Результат
==	дорівнює	x == 5	True
!=	не дорівнює	x != y	True
>	більше	x > y	True
<	менше	x < y	False
>=	більше або дорівнює	x >= 5	True
<=	менше або дорівнює	x <= 4	False

Важливо: Не плутайте оператор **присвоєння** = (ми кладемо значення в змінну) з оператором **порівняння** == (ми перевіряємо, чи дорівнюють значення одне одному). Це одна з найпоширеніших помилок-друкарських помилок (typo) у програмуванні!

```
age = 20
is_adult = age >= 18 # is_adult стане True
print(is_adult)      # True

name = "Олена"
print(name == "Іван") # False
```

Оператори порівняння працюють не тільки з числами. Можна порівнювати рядки (вони порівнюються за алфавітом, лексикографічно), але з цим треба бути обережним через регістр ("А" < "а"). Також можна порівнювати різні типи даних? У Python 3 це призведе до помилки, що є безпечнішим підходом.

Логічні оператори

Що робити, якщо умова має бути складною? Наприклад, "піца смачна і гаряча", або "клієнт замовив десерт **або** напій". Для цього існують логічні оператори: and, or, not.

1. and (логічне "І"): Повертає True тільки тоді, коли **обидва** вирази (і ліворуч, і праворуч) є True. В усіх інших випадках – False.

```
temperature = 70
time = 12
# Перевіряємо, чи піца гаряча (більше 60) І чи час обідній (більше 11)
is_hot_and_lunch = (temperature > 60) and (time > 11)
print(is_hot_and_lunch) # True (70>60 = True, 12>11 = True) -> True
```

2. or (логічне "АБО"): Повертає True, якщо **хоча б один** з виразів є True. Повертає False тільки тоді, коли обидва вирази – False.

```
has_dessert = False
has_drink = True
# Перевіряємо, чи замовив клієнт щось додатково (десерт АБО напій)
has_extra = has_dessert or has_drink
print(has_extra) # True (False or True = True)
```

3. not (логічне "НЕ"): Це унарний оператор (застосовується до одного виразу). Він просто "перевертає" булеве значення: True перетворює на False, а False на True.

```
is_holiday = False
# Перевіряємо, чи сьогодні робочий день (НЕ вихідний)
is_working_day = not is_holiday
print(is_working_day) # True
```

Таблиці істинності

Для кращого розуміння роботи логічних операторів використовують таблиці істинності.

A	B	A and B	A or B	not A
True	True	True	True	False

A	B	A and B	A or B	n ot A
T rue	F alse	Fal se	T rue	Fa lse
F alse	T rue	Fal se	T rue	Tr ue
F alse	F alse	Fal se	F alse	Tr ue

Зверніть увагу на рядок, де A=True, B=False. and дає False (бо обидва не True), а or дає True (бо один з них True).

Ці оператори є цеглинками, з яких будується будь-яка логіка програми. Наприклад, перевірка, чи може користувач отримати знижку:

```
user_age = 65
is_student = False
is_weekend = True
```

```
# Знижка для пенсіонерів (вік > 60) АБО для студентів (is_student ==
True), але НЕ у вихідні
discount_available = (user_age > 60 or is_student) and not is_weekend
print(discount_available) # (True or False) and not True = True and False =
False
```

Наступним кроком ми навчимося використовувати ці логічні вирази в конструкціях if, щоб програма могла виконувати різні дії в залежності від результату.

3. Присвоєння з операціями та спеціальні оператори (in, is)

У програмуванні дуже часто доводиться змінювати значення змінної на основі її поточного значення. Наприклад, додати щось до кошика покупок, збільшити лічильник переглядів або змінити статус користувача. Для таких операцій Python пропонує не лише звичайний оператор присвоєння =, але й комбіновані оператори, які роблять код коротшим і зрозумілішим.

Комбіновані оператори присвоєння (Assignment operators)

Ідея проста: замість того щоб писати $x = x + 5$, ми можемо написати $x += 5$. Це означає "візьми поточне значення x , додай до нього 5 і результат поклади назад у x ".

Розглянемо всі варіанти на прикладі ігрового персонажа:

У нашого героя є здоров'я, досвід та золото

`health = 100`

`experience = 0`

`gold = 50`

1. Додавання з присвоєнням (+=)

`health += 20` *# герой знайшов аптечку*

`print(health)` *# 120 (health = health + 20)*

2. Віднімання з присвоєнням (-=)

`health -= 30` *# герой отримав поранення*

`print(health)` *# 90 (health = health - 30)*

3. Множення з присвоєнням (=)*

`experience *= 2` *# герой отримав подвійний досвід за вбивство боса*

`print(experience)` *# 0 (0 * 2 = 0 – зверніть увагу!)*

4. Ділення з присвоєнням (/=)

`gold /= 2` *# герой заплатив половину золота за інформацію*

`print(gold)` *# 25.0 (зверніть увагу: результат завжди float)*

5. Цілочисельне ділення з присвоєнням (//=)

`gold = 50`

`gold //= 3` *# герой поділив 50 золотих на 3 частини (тільки цілі!)*

`print(gold)` *# 16 (50 // 3 = 16)*

6. Остача від ділення з присвоєнням (%=)

`gold = 50`

`gold %= 3` *# скільки золотих залишиться після розподілу на 3?*

`print(gold)` *# 2 (50 % 3 = 2)*

*# 7. Піднесення до степеня з присвоєнням (**=)*

```
level = 2
```

```
level **= 3 # рівень персонажа піднісся в куб
```

```
print(level) # 8
```

Такі оператори не тільки скорочують код, але й роблять його більш ідіоматичним (зрозумілим для досвідчених Python-програмістів). Вони працюють з будь-якими числовими типами і часто використовуються в циклах, про які ми поговоримо пізніше.

Спеціальні оператори Python

Окрім математичних операторів, Python має кілька унікальних операторів, які роблять роботу з даними дуже зручною.

Оператор членства in (і not in)

Цей оператор перевіряє, чи є елемент у певній послідовності (наприклад, у рядку, списку, кортежі). Він повертає True, якщо елемент знайдено, і False – якщо ні.

Перевірка наявності підрядка в рядку

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

```
if "@" in email:
```

```
    print("Email містить символ @") # Це буде виведено
```

Перевірка наявності елемента в списку

```
allowed_colors = ["червоний", "зелений", "синій"]
```

```
user_color = "жовтий"
```

```
if user_color not in allowed_colors:
```

```
    print(f"Колір '{user_color}' не дозволений. Оберіть зі списку.")
```

Перевірка ключа в словнику (дуже часто використовується!)

```
user = {"name": "Олена", "age": 19, "city": "Київ"}
```

```
if "age" in user:
```

```
    print(f"Вік користувача: {user['age']}")
```

Оператор in робить код неймовірно чистим і зрозумілим. Уявіть, як би це виглядало без нього – довелося б писати цикл і вручну порівнювати кожен елемент.

Оператор тотожності is (і is not)

Тут часто виникає плутанина у початківців. `==` перевіряє **рівність значень** (чи дорівнює вміст однієї змінної вмісту іншої). `is` перевіряє **тотожність об'єктів** (чи є дві змінні посиланням на один і той самий об'єкт у пам'яті).

Приклад 1: Робота з простими типами

```
a = [1, 2, 3]
```

```
b = [1, 2, 3]
```

```
c = a
```

```
print(a == b) # True (значення однакові)
```

```
print(a is b) # False (це різні списки в пам'яті)
```

```
print(a is c) # True (c посилається на той самий об'єкт, що й a)
```

Приклад 2: Особливість з малими числами (інтернування)

```
x = 256
```

```
y = 256
```

```
print(x is y) # True (Python кешує малі числа від -5 до 256)
```

```
x = 1000
```

```
y = 1000
```

```
print(x is y) # False (великі числа не кешуються)
```

Приклад 3: Головне використання is – порівняння з None

```
result = None
```

```
if result is None: # Так РЕКОМЕНДУЄТЬСЯ робити за PEP 8
```

```
    print("Результату немає")
```

```
if result == None: # Так технічно працює, але не рекомендується
```

```
    print("Результату немає")
```

Чому важливо знати різницю?

Уявіть, що ви пишете програму для банку. Два різних рахунки можуть мати однакову суму грошей (`==` буде `True`), але це різні рахунки (`is` буде `False`). Також запам'ятайте головне правило: `None` завжди перевіряють через `is`.

4. Введення та виведення даних: спілкування з користувачем

Досі наші програми були трохи "самодостатніми" – всі дані ми прописували прямо в коді. Але справжня програма має взаємодіяти з користувачем: отримувати від нього дані, обробляти їх і виводити результати. У Python за це відповідають дві базові функції: `print()` для виведення та `input()` для введення.

Функція `print()`: мистецтво виведення

З `print()` ми вже знайомі, але вона вміє набагато більше, ніж просто друкувати один рядок.

Можна вивести кілька значень через кому

```
name = "Максим"
```

```
age = 20
```

```
print("Ім'я:", name, "Вік:", age) # Ім'я: Максим Вік: 20
```

За замовчуванням між значеннями вставляється пробіл

Параметр `sep` (separator) дозволяє змінити розділювач

```
print("Максим", "Олена", "Іван", sep=" | ") # Максим | Олена | Іван
```

Параметр `end` дозволяє змінити те, що друкується в кінці (замість `\n` - переходу на новий рядок)

```
print("Завантаження", end="")
```

```
print(".", end="")
```

```
print(".", end="")
```

```
print(".") # Виведе: Завантаження....
```

Форматування рядків: f-strings

Раніше програмісти використовували метод `format()` або оператор `%`, але сучасний Python (починаючи з версії 3.6) пропонує найкращий спосіб – **f-strings**. Це неймовірно зручно та читабельно.

```
name = "Анна"
```

```
city = "Львів"
```

```
temperature = 18.5
```

Старий спосіб (не використовуйте його без потреби)

```
print("Мене звати {}, я з міста {}".format(name, city))
```

СУЧАСНИЙ СПОСІБ (f-strings)

```
print(f'Мене звати {name}, я з міста {city}')
```

У f-strings можна виконувати обчислення прямо всередині {}

```
price = 120
```

```
discount = 15
```

```
print(f'Ціна зі знижкою: {price - discount} грн')
```

```
print(f'Ціна з ПДВ: {price * 1.2:.2f} грн') # :.2f - два знаки після коми
```

Можна використовувати багаторядкові f-strings

```
user_info = f"""
```

```
--- Профіль користувача ---
```

```
Ім'я: {name}
```

```
Місто: {city}
```

```
Температура у місті: {temperature}°C
```

```
-----
```

```
"""
```

```
print(user_info)
```

Функція input(): слухаємо користувача

Функція input() призупиняє виконання програми і чекає, поки користувач введе щось з клавіатури і натисне Enter. Все, що ввів користувач, повертається у вигляді **рядка (str)**.

Найпростіше введення

```
name = input("Як вас звати? ")
```

```
print(f'Приємно познайомитись, {name}!')
```

Найважливіше правило: input() завжди повертає рядок. Якщо вам потрібне число, його треба явно перетворити.

Неправильно (буде помилка)

```
age = input("Скільки вам років? ")
```

print(f'Наступного року вам буде {age + 1}') # Помилка! Не можна додати число до рядка

Правильно

```
age = int(input("Скільки вам років? "))
```

```
print(f'Наступного року вам буде {age + 1}')
```

```
# Можна і з float  
price = float(input("Введіть ціну товару: "))  
print(f"Ціна з ПДВ: {price * 1.2:.2f}")
```

Обробка введених даних

Часто потрібно отримати від користувача кілька значень. Ось кілька корисних прийомів:

```
# Спосіб 1: Послідовні input()  
name = input("Ім'я: ")  
age = int(input("Вік: "))  
city = input("Місто: ")
```

```
# Спосіб 2: Розділення рядка (якщо користувач вводить все в один рядок)  
# Наприклад, для введення координат "10 20 30"  
data = input("Введіть три числа через пробіл: ")  
numbers = data.split() # розбиває рядок на список слів  
x, y, z = map(int, numbers) # перетворює кожен елемент на int  
print(f"Сума: {x + y + z}")
```

```
# Спосіб 3 (складніший, але елегантний):  
x, y, z = map(int, input("Введіть три числа: ").split())  
print(f"Сума: {x + y + z}")
```

Спеціальні символи (escape sequences)

Іноді нам потрібно вивести спеціальні символи, які важко надрукувати звичайним способом. Для цього використовують ескапе-послідовності, які починаються зі зворотнього слеша \.

```
# \n - новий рядок  
print("Перший рядок\nДругий рядок")
```

```
# \t - табуляція (відступ)  
print("Ім'я:\tАнна")  
print("Вік:\t20")
```

```
# \\ - вивести сам зворотній слеш
```



```
print("Шлях до файлу: C:\\Users\\Name\\Desktop")
```

\" або \' - якщо потрібно вивести лапки всередині рядка

```
print("Він сказав: \"Привіт!\"")
```

Але є простіший спосіб: чергувати типи лапок

```
print('Він сказав: "Привіт!"')
```

Практичний приклад: простий калькулятор

Об'єднаємо все, що ми вивчили в цій лекції, в невелику корисну програму:

Програма-калькулятор для піцерії

```
print("=" * 40)
```

```
print("🍷 КАЛЬКУЛЯТОР ПІЦИ 🍷")
```

```
print("=" * 40)
```

Введення даних

```
pizza_name = input("Назва піци: ")
```

```
price = float(input("Ціна за шматок (грн): "))
```

```
slices = int(input("Кількість шматків: "))
```

```
people = int(input("Кількість людей: "))
```

Обчислення

```
total_cost = price * slices
```

```
cost_per_person = total_cost / people
```

```
slices_per_person = slices // people
```

```
leftover_slices = slices % people
```

Виведення результату (з використанням f-strings)

```
print("\n" + "=" * 40)
```

```
print("📋 РЕЗУЛЬТАТ:")
```

```
print(f"Піца: {pizza_name}")
```

```
print(f"Загальна вартість: {total_cost:.2f} грн")
```

```
print(f"Кожен платить: {cost_per_person:.2f} грн")
```

```
print(f"Шматків на кожного: {slices_per_person}")
```

```
print(f"Залишиться шматків: {leftover_slices}")
```

Перевірка, чи всі отримають порівну (використовуємо логічний оператор)

```
if leftover_slices == 0:
    print("✓ Піца поділилась ідеально!")
else:
    print("⚠ Доведеться ділити останні шматочки...")
print("=" * 40)
```

Ця програма демонструє практично все, що ми вивчили: введення даних (input()), перетворення типів (int(), float()), обчислення (арифметичні оператори), логічні перевірки (if, ==) та красиве виведення (f-strings, множення рядків). Наступним кроком буде вивчення більш гнучких способів прийняття рішень – конструкцій розгалуження.

Питання для обговорення

1. Поясніть різницю між операторами / та //. Наведіть приклад, коли використання // є більш доречним, ніж звичайне ділення.

2. Чому результатом виразу 10 / 2 буде 5.0, а не 5? Як це пов'язано з типами даних у Python?

3. Яка помилка в коді: if user_age = 18:? Чому використання одного знаку дорівнює в умові є логічною помилкою, а не просто синтаксичною?

4. Поясніть принцип роботи логічного оператора and за допомогою таблиці істинності. Наведіть приклад складної умови з реального життя, яка використовує and (наприклад, для перевірки, чи можна дивитися фільм).

5. Що таке "коротке замикання" (short-circuit evaluation) у логічних операторах? Чим воно корисне і як може запобігти помилкам (наприклад, діленню на нуль)?

6. У чому різниця між операторами == та is? Чи можна їх використовувати як взаємозамінні? Коли варто використовувати is?

7. Поясніть різницю між параметрами sep та end у функції print(). Наведіть приклад, де зміна цих параметрів робить виведення більш естетичним або зручним.

8. Чому після використання input() потрібно часто застосовувати функції int() або float()? Що станеться, якщо спробувати додати число до результату input() без перетворення?

9. Напишіть програму, яка запитує у користувача ім'я, вік та місто, а потім виводить привітання з використанням f-рядка. Чому f-рядки вважаються сучасним і кращим способом форматування?

10. Що таке escape-послідовності? Для чого використовуються `\n` та `\t`? Як вивести на екран сам символ зворотнього слеша (`\`)?

ЛЕКЦІЯ 3

Умовні конструкції та розгалуження програми

План лекції

1. Концепція розгалуження та простий if: як програма обирає шлях.

Тут ми пояснимо саму ідею розгалуження в програмуванні, розберемо синтаксис простого if, і найголовніше – детально зупинимося на унікальній особливості Python: відступи як спосіб визначення блоків коду. Це критично важливо для початківців, щоб зрозуміти структуру програм.

2. Бінарний вибір: if-else та множинне розгалуження з elif.

Навчимо програму обробляти дві альтернативи ("так-ні") та, що ще важливіше, багато варіантів вибору за допомогою конструкції if-elif-else. Розберемо, як правильно вибудовувати ланцюжки умов і чому це краще за вкладені конструкції.

3. Тернарний оператор та вкладені умови: компактність чи читабельність?

Познайомимося зі скороченим способом запису простих розгалужень – тернарним оператором. Також розглянемо ситуації, коли умови знаходяться всередині інших умов, і обговоримо, як не заплутатися у вкладених структурах.

4. Практичні аспекти та типові помилки: перевірка на None, порожнечу та оптимізація.

Приділимо увагу хорошим практикам: як правильно порівнювати з None (згідно з PEP 8), як перевіряти, чи список порожній, і що таке "коротке замикання" (short-circuit evaluation) – корисна особливість логічних операторів, яка може впливати на ефективність коду.

1. Концепція розгалуження та простий if: як програма обирає шлях

Уявіть собі звичайний ранок. Якщо на вулиці йде дощ, ви берете парасольку. Якщо світить сонце – вдягаєте сонцезахисні окуляри. Якщо температура нижче нуля – берете теплу куртку. Ваші дії залежать від умов. Саме так само працюють і комп'ютерні програми: вони аналізують поточний стан даних і виконують різні дії залежно від результату цього аналізу. Це називається **розгалуженням**.

До цього моменту всі наші програми були **лінійними** – вони виконували команди одну за одною, від першого рядка до останнього, незалежно ні від чого. Але справжня потужність програмування розкривається саме тоді, коли ми додаємо можливість вибору.

Конструкція if: перший крок до "розумної" програми

Найпростіша форма розгалуження в Python – це конструкція if (з англійської "якщо"). Вона працює за дуже простою логікою: "Якщо умова є істинною, виконай цей блок коду".

Синтаксис виглядає так:

if умова:

блок коду, який виконається, якщо умова True

команда1

команда2

...

решта програми (виконується завжди, незалежно від умови)

Розглянемо реальний приклад. Уявімо, що ми створюємо програму для перевірки віку користувача на сайті з обмеженим доступом:

```
age = 17
```

```
if age < 18:
```

```
    print("Доступ заборонено! Вам немає 18 років.")
```

```
    print("Будь ласка, залиште цю сторінку.")
```

```
print("Дякуємо за використання нашого сайту.")
```

Якщо age дорівнює 17, умова age < 18 буде True, і ми побачимо:

Доступ заборонено! Вам немає 18 років.

Будь ласка, залиште цю сторінку.

Дякуємо за використання нашого сайту.

Якщо ж age дорівнює 20, умова буде False, і програма просто пропустить блок if:

Дякуємо за використання нашого сайту.

Найважливіше правило Python: відступи визначають структуру!

Зверніть увагу на те, як записаний код. У багатьох інших мовах програмування (C++, Java, JavaScript) блоки коду виділяються фігурними дужками {}. У Python цю роль виконують **відступи**. Це унікальна особливість мови, яка спочатку може здаватися незвичною, але насправді робить код неймовірно чистим і читабельним.

Правила відступів у Python:

1. Всі інструкції, які належать до одного блоку (наприклад, до блоку if), повинні мати **однаковий відступ**.

2. Стандартом (PEP 8) рекомендується використовувати **4 пробіли** для одного рівня відступу. Будь-яка сучасна IDE (VS Code, PyCharm) робить це автоматично, коли ви натискаєте Tab.

3. Відступ має сенс – він показує, які команди "підпорядковані" умові.

Це означає, що в Python відступи – це не просто прикраса, а частина синтаксису. Якщо ви зробите неправильний відступ, програма не просто виглядатиме негарно – вона не працюватиме або працюватиме не так, як ви очікуєте.

ПРАВИЛЬНО

```
x = 10
if x > 5:
    print("x більше 5") # 4 пробіли перед print
    print("Це теж частина блоку if")
```

НЕПРАВИЛЬНО (IndentationError)

```
x = 10
if x > 5:
print("x більше 5") # Помилка! Немає відступу
```

Що може бути умовою?

Умовою в if може бути будь-який вираз, який повертає булеве значення (True або False). Це можуть бути:

- Оператори порівняння (>, <, ==, !=, >=, <=)
- Логічні оператори (and, or, not)
- Результат роботи функцій, що повертають bool
- Навіть просто змінна булевого типу

```
is_logged_in = True
if is_logged_in:
    print("Ласкаво просимо до особистого кабінету!")
```

Цікавий момент: Python дозволяє використовувати в умові не тільки булеві значення. Будь-яке значення може бути інтерпретоване як True або False. Це називається "truthy" та "falsy". Наприклад:

- Число 0 вважається False, всі інші числа – True

- Порожній рядок "" вважається False, непорожній – True
- Порожні списки, словники – False
- None завжди False

```
name = "" # порожній рядок
if name:
    print(f"Привіт, {name}!")
else:
    print("Ви не ввели ім'я!") # Виконається цей блок
```

Це дуже зручно для швидких перевірок, але початківцям варто бути з цим обережними, щоб не виникало плутанини.

2. Бінарний вибір: if-else та множинне розгалуження з elif

Конструкція if сама по собі корисна, але часто нам потрібно обробити дві альтернативи: "якщо умова виконується – зроби одне, інакше – зроби інше". Для цього існує конструкція if-else.

Конструкція if-else: дві дороги

Синтаксис дуже простий:

```
if умова:
    # блок коду, якщо умова True
    команда_так
else:
    # блок коду, якщо умова False
    команда_ні
```

Повернемося до нашого прикладу з віком:

```
age = 20

if age >= 18:
    print("Доступ дозволено! Ласкаво просимо.")
    print("Бажаєте придбати преміум-доступ?")
else:
    print("Доступ заборонено! Вам немає 18 років.")
    print("Повертайтеся, коли виростете.")
```



```
print("Дякуємо за візит.")
```

Тепер програма завжди виконає один із двох блоків, залежно від значення `age`. Це і є **бінарне розгалуження** – вибір з двох можливих шляхів.

Важливо: блок `else` не може існувати сам по собі, без попереднього `if`. Він завжди прив'язаний до найближчого `if`.

Множинне розгалуження: `if-elif-else`

А що робити, якщо у нас не два, а три, чотири або більше варіантів? Наприклад, ми хочемо поставити оцінку студенту залежно від його балу:

- 90+ балів: "Відмінно"
- 75-89 балів: "Добре"
- 60-74 бали: "Задовільно"
- менше 60: "Незадовільно"

Можна було б написати багато вкладених `if-else`, але це швидко стає заплутаним. На щастя, Python пропонує елегантне рішення – конструкцію `if-elif-else`. `elif` – це скорочення від "else if" (інакше якщо).

```
score = 82
```

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

```
print(f"Ваша оцінка: {grade}")
```

Як це працює?

1. Спочатку перевіряється умова `score >= 90`. Якщо вона `True`, виконується відповідний блок, і решта `elif` та `else` ігноруються.

2. Якщо перша умова `False`, перевіряється наступна (`score >= 75`).

3. Якщо вона `True`, виконується її блок, і далі перевірки не відбуваються.

4. Якщо жодна з умов не виявилася `True`, виконується блок `else` (якщо він є).

Ключовий момент: порядок умов має значення! Якби ми написали спочатку `score >= 60`, то всі студенти з балами 60+ потрапили б туди, і до наступних умов справа б не дійшла. Тому в таких ланцюжках умови зазвичай розташовують від найвимогливішої до найменш вимогливої.

Скільки може бути `elif`?

Технічно – скільки завгодно. Але якщо їх стає дуже багато (більше 5-7), варто задуматися, чи не можна спростити логіку, можливо, використавши словник або інші структури даних.

Чи обов'язковий `else`?

Ні, `else` не обов'язковий. Можна мати конструкцію з кількома `elif` без завершального `else`. В такому випадку, якщо жодна умова не спрацює, просто нічого не виконається.

```
command = "start"
```

```
if command == "start":
    print("Запускаємо програму...")
elif command == "stop":
    print("Зупиняємо програму...")
elif command == "pause":
    print("Пауза...")
```

Якщо команда невідома, просто нічого не робимо

Практичний приклад: програма-консультант

Об'єднаємо все вивчене в невелику, але корисну програму, яка допомагає обрати одяг за погодою:

```
print("❁ ПОРАДНИК ОДЯГУ ❁")
print("-" * 30)
```

Отримуємо дані від користувача

```
temperature = float(input("Яка температура повітря сьогодні (°C)? "))
is_raining_input = input("Чи йде дощ? (так/ні): ")
```

Перетворюємо відповідь про дощ на булеве значення

```
if is_raining_input.lower() == "так":
```

```

is_raining = True
else:
    is_raining = False

# Даємо пораду на основі температури
print("\n🔍 МОЯ РЕКОМЕНДАЦІЯ:")
if temperature >= 25:
    print("▶ Футболка та шорти")
    if is_raining:
        print("▶ Але не забудьте парасольку!") # Вкладений if
elif temperature >= 15:
    print("▶ Легка кофта або светр")
    if is_raining:
        print("▶ Візьміть куртку з капюшоном")
elif temperature >= 5:
    print("▶ Тепла куртка та шапка")
else:
    print("▶ Пуховик, шарф і рукавички! Надворі дуже холодно")

# Додаткова перевірка на особливі умови
if temperature < 0 and is_raining:
    print("⚠ УВАГА: Можливий ожеледь! Будьте обережні!")

print("\n🌞 Гарного дня! ☀")

```

Ця програма демонструє:

- Використання if-elif-else для множинного вибору
- Вкладений if всередині блоків
- Перетворення даних користувача на булеві значення
- Комбінування умов через логічні оператори (and)

У наступних пунктах ми розглянемо ще більш гнучкі способи роботи з умовами, зокрема тернарний оператор для компактного запису простих розгалужень, та важливі практичні нюанси, які допоможуть писати надійний код.

3. Тернарний оператор та вкладені умови: компактність чи читабельність?

Коли ми працюємо з умовами, іноді виникає спокуса написати все якомога компактніше, а іноді – навпаки, розгалудити логіку в глибокі структури. У цьому пункті ми розглянемо два протилежних підходи: тернарний оператор для стислого запису простих умов та вкладені конструкції для складних багаторівневих рішень. Головне питання, яке ми маємо навчитися вирішувати – коли що застосовувати.

Тернарний оператор (умовний вираз)

Часто буває ситуація, коли нам потрібно присвоїти змінній одне з двох значень залежно від умови. Звичайний if-else для цього може бути надто громіздким:

Звичайний спосіб

```
age = 20
if age >= 18:
    status = "дорослий"
else:
    status = "неповнолітній"
```

Для таких простих випадків у Python є спеціальна конструкція – **тернарний оператор** (також відомий як умовний вираз). Його синтаксис:

змінна = значення_якщо_True if умова else значення_якщо_False
Той самий приклад із віком виглядатиме так:

```
age = 20
status = "дорослий" if age >= 18 else "неповнолітній"
print(status) # дорослий
```

Це набагато компактніше! Але важливо розуміти: тернарний оператор – це саме **вираз**, який повертає значення, а не інструкція. Тому його можна використовувати не тільки для присвоєння, а й у інших місцях:

Усередині f-рядка

```
age = 17
print(f"Ви { 'допущені' if age >= 18 else 'не допущені' } до голосування")
```

У списку

```
numbers = [1, 2, 3, 4, 5]
parity = ["парне" if x % 2 == 0 else "непарне" for x in numbers]
```

```
print(parity) # ['непарне', 'парне', 'непарне', 'парне', 'непарне']
```

Коли варто використовувати тернарний оператор?

Тернарний оператор чудово підходить для простих, лінійних умов, де є лише два варіанти і вони короткі. Він робить код стислим і зрозумілим.

Коли НЕ варто використовувати тернарний оператор?

1. Якщо умова складна (з and/or).
2. Якщо значення, що повертаються, є довгими виразами.
3. Якщо ви намагаєтесь вкласти один тернарний оператор в інший (це майже завжди погана ідея).

Поганий приклад (дуже важко читати)

```
category = "A" if score >= 90 else ("B" if score >= 80 else ("C" if score >= 70 else "D"))
```

Краще використати звичайний if-elif-else для такої логіки

Вкладені умовні оператори

Іноді логіка програми вимагає, щоб одна умова знаходилась всередині іншої. Наприклад, ми перевіряємо, чи є у користувача доступ до системи, і якщо так – перевіряємо його роль. Це називають **вкладеними умовами** (nested conditions).

```
user_authenticated = True
```

```
user_role = "admin"
```

```
if user_authenticated:
```

```
    print("Користувач автентифікований")
```

```
    if user_role == "admin":
```

```
        print("Доступ до адмін-панелі дозволено")
```

```
    elif user_role == "manager":
```

```
        print("Доступ до звітів дозволено")
```

```
    else:
```

```
        print("Доступ до звичайного інтерфейсу")
```

```
else:
```

```
    print("Будь ласка, увійдіть у систему")
```

Така структура цілком логічна: зовнішній if перевіряє головну умову, а внутрішній – уточнює деталі.

Проблема глибокої вкладеності

Проте з вкладеними умовами легко переборщити. Уявіть собі такий код:

Приклад "драбинки жаху"

```
if condition1:
    if condition2:
        if condition3:
            if condition4:
                # зроби щось
            else:
                # обробка
        else:
            # обробка
    else:
        # обробка
else:
    # обробка
```

Такий код дуже важко читати. Він нагадує драбину, яка веде в нікуди. Якщо ви зустрічаєте 3-4 рівні вкладеності – це сигнал, що логіку варто перебудувати.

Як уникнути глибокої вкладеності?

Існує кілька прийомів:

1. **Раннє повернення (early return)** – якщо умова не виконується, одразу виходимо з функції (ми розглянемо це детальніше, коли вивчатимемо функції).
2. **Об'єднання умов через and.**
3. **Використання словників** замість довгих if-elif-else ланцюжків.

Замість глибокої вкладеності

```
if user_authenticated:
    if user_role == "admin" and user_active:
        print("Адмін доступ")
```

Краще об'єднати умови

```
if user_authenticated and user_role == "admin" and user_active:
    print("Адмін доступ")
```

4. Практичні аспекти та типові помилки: перевірка на None, порожнечу та оптимізація

Тепер, коли ми знаємо основні інструменти роботи з умовами, настав час поговорити про професійні "секрети" – як писати не просто правильний, а гарний, ефективний та надійний код.

Порівняння з None: рекомендації PEP 8

None – це спеціальне значення в Python, яке означає "відсутність значення" або "нічого". Воно використовується дуже часто: функції, які нічого не повертають, насправді повертають None; змінні, які ще не отримали значення, часто ініціалізують як None.

Як правильно перевіряти, чи змінна дорівнює None?

```
result = None
```

```
# ПРАВИЛЬНО (PEP 8)
```

```
if result is None:  
    print("Результату немає")
```

```
if result is not None:  
    print(f"Результат: {result}")
```

```
# НЕПРАВИЛЬНО (хоча технічно працює)
```

```
if result == None:  
    print("Цього краще уникати")
```

Чому is None, а не == None? Тому що None у Python існує в єдиному екземплярі (це так званий "синглтон"). Оператор is перевіряє саме тотожність об'єктів, що для None гарантовано працює правильно. Оператор == перевіряє рівність значень, і хтось міг би "перевизначити" його для свого класу, що призвело б до неочікуваної поведінки. Тому в PEP 8 (офіційному посібнику зі стилю коду Python) чітко сказано: порівнюйте з None тільки через is або is not.

Перевірка на порожні колекції

Дуже часто потрібно перевірити, чи список, рядок, словник або інша колекція є порожніми. Новачки часто пишуть так:

```
my_list = []
```

Спосіб новачка (НЕ РЕКОМЕНДУЄТЬСЯ)

```
if len(my_list) > 0:  
    print("Список не порожній")
```

```
if len(my_list) == 0:  
    print("Список порожній")
```

Але в Python є більш елегантний та "pythonic" спосіб. Пам'ятаєте, ми говорили про "truthy" та "falsy" значення? Порожні колекції вважаються False в логічному контексті.

```
my_list = []
```

ПРАВИЛЬНО (pythonic way)

```
if my_list: # якщо список НЕ порожній  
    print("Список не порожній")
```

```
if not my_list: # якщо список порожній  
    print("Список порожній")
```

Те саме працює з рядками, словниками, кортежами, множинами

```
my_string = ""  
if not my_string:  
    print("Рядок порожній")
```

```
my_dict = {}
```

```
if my_dict:  
    print("Словник не порожній") # цей рядок не виконається
```

Цей підхід робить код більш природним і легшим для читання.

Коротке замикання (Short-circuit evaluation)

Логічні оператори `and` та `or` в Python мають цікаву особливість: вони обчислюються за лінивим принципом. Це означає, що Python припиняє обчислення виразу, щойно результат стає відомим. Це називається **коротким замиканням** (short-circuit evaluation).

Як це працює:

- Для `and`: якщо лівий операнд `False`, то правий вже не обчислюється, бо результат всього виразу точно буде `False`.

- Для `or`: якщо лівий операнд `True`, то правий вже не обчислюється, бо результат всього виразу точно буде `True`.

Приклад 1: and

`x = 5`

`y = 0`

Тут права частина (x / y) ніколи не обчислиться, бо y == 0 дає False

`if y != 0 and x / y > 2:`

`print("Це не виконається, але помилки ділення на нуль не буде!")`

Приклад 2: or

`name = input("Введіть ім'я: ") or "Анонім"`

Якщо користувач введе порожній рядок (False), то змінна name отримає значення "Анонім"

`print(f"Привіт, {name}!")`

Це дуже корисна особливість, яку варто використовувати:

- Для запобігання помилкам (як у прикладі з діленням)
- Для надання значень за замовчуванням (як у прикладі з `or`)
- Для оптимізації – якщо ліва частина умови обчислюється швидко, а права – повільно, краще поставити швидку перевірку зліва

Типові помилки при роботі з умовами

1. **Плутанина між `==` та `==`** – найпоширеніша помилка початківців.

`x = 10`

`if x = 5: # ПОМИЛКА! Тут має бути ==`

`print("Це не спрацює")`

2. **Забутий двокрапка** : в кінці рядка з `if`, `elif`, `else`.

`if x > 5 # ПОМИЛКА! Пропущено двокрапку`

`print("x більше 5")`

3. **Неправильний порядок умов у `if-elif`**.

`x = 85`

`if x >= 60: # Ця умова спрацює першою`

`grade = "Задовільно"`

`elif x >= 75:`

```

    grade = "Добре" # Ніколи не виконається!
elif x >= 90:
    grade = "Відмінно" # Ніколи не виконається!
4. Занадто складна умова.

```

```

# Замість цього
if (user.is_active and user.has_permission and not user.is_blocked) or
(user.is_admin and not user.is_blocked):
    # ...

# Краще винести в окрему змінну
can_access = (user.is_active and user.has_permission and not user.is_blocked)
or (user.is_admin and not user.is_blocked)
if can_access:
    # ...

```

Практичний приклад: вдосконалена програма реєстрації

Об'єднаємо всі вивчені концепції в програму для перевірки даних користувача при реєстрації:

```

print("👋 ФОРМА РЕЄСТРАЦІЇ")
print("=" * 40)

# Отримуємо дані
username = input("Придумайте ім'я користувача: ")
email = input("Введіть email: ")
password = input("Введіть пароль: ")
age = input("Введіть ваш вік: ")

# Валідація даних з використанням короткого замикання
errors = []

# Перевірка імені (не порожнє, мінімум 3 символи)
if not username:
    errors.append("Ім'я користувача не може бути порожнім")
elif len(username) < 3:
    errors.append("Ім'я користувача має містити хоча б 3 символи")

```

```

# Перевірка email (має містити @, проста перевірка)
if not email:
    errors.append("Email не може бути порожнім")
elif "@" not in email: # використовуємо оператор in
    errors.append("Email має містити символ @")

# Перевірка пароля (не порожній, мінімум 6 символів)
if not password:
    errors.append("Пароль не може бути порожнім")
elif len(password) < 6:
    errors.append("Пароль має містити хоча б 6 символів")

# Перевірка віку (має бути числом і не менше 13)
if not age:
    errors.append("Вік не може бути порожнім")
else:
    # Тернарний оператор для визначення, чи age є числом
    age = int(age) if age.isdigit() else None
    if age is None: # перевірка на None через is
        errors.append("Вік має бути числом")
    elif age < 13:
        errors.append("Вам має бути щонайменше 13 років")

# Виведення результату
print("\n" + "=" * 40)
if not errors: # якщо список помилок порожній (falsy)
    print("✓ РЕЄСТРАЦІЯ УСПІШНА!")
    print(f"Ласкаво просимо, {username}!")
    # Тернарний оператор для привітання
    welcome_message = "Бажаєте отримувати новини?" if age and age >= 18
else "Ласкаво просимо до спільноти!"
    print(welcome_message)
else:
    print("✗ ПОМИЛКИ РЕЄСТРАЦІЇ:")
    for error in errors:
        print(f"• {error}")

```

```
print("=" * 40)
```

Ця програма демонструє професійний підхід:

- Використання "falsy" для перевірки порожніх значень (if not username)
- Перевірку на None через is
- Тернарний оператор для присвоєння
- Коротке замикання (неявно, коли ми перевіряємо age.isdigit()) тільки якщо age не порожнє)

- Збір помилок у список замість негайного виходу

У наступній лекції ми перейдемо до циклів – ще одного фундаментального інструменту, який дозволяє виконувати повторювані дії автоматично.

Питання для обговорення

1. Чому в Python важливі відступи, а не фігурні дужки? Як ви вважаєте, це полегшує читання коду чи створює додаткові труднощі для програміста?
2. Проаналізуйте код. Чи є в ньому помилка? Якщо так, то яка?

```
x = 10
if x > 5:
    print("x більше 5")
    print("Це також частина if")
print("Це поза if")
```

3. У чому різниця між конструкціями if-elif-else та кількома окремими if? Чому для перевірки взаємовиключних умов краще використовувати саме elif?
4. Напишіть умовну конструкцію, яка перевіряє, чи число є додатним, від'ємним або нулем. Використовуйте if-elif-else.
5. Що таке тернарний оператор? Перепишіть наступний код з використанням тернарного оператора:

```
if age >= 18:
    status = "дорослий"
else:
    status = "неповнолітній"
```

6. У чому небезпека глибоко вкладених умов? Запропонуйте спосіб, як переписати код, щоб уникнути "драбини" з трьома і більше рівнями вкладеності.

7. Чому за рекомендацією PEP 8 для порівняння з None слід використовувати is None або is not None, а не == None? Яка технічна різниця між цими операціями?

8. Поясніть, як Python інтерпретує різні значення як True або False (truthy та falsy). Чому код if not my_list: працює коректно, якщо my_list порожній?

9. Що буде результатом виконання наступного коду і чому? Поясніть з точки зору пріоритету операцій та короткого замикання.

```
x = 10
y = 0
if y != 0 and x / y > 2:
    print("Умова виконана")
else:
    print("Умова не виконана")
```

10. Напишіть програму, яка запитує у користувача оцінку (від 0 до 100) і виводить літерну оцінку (A, B, C, D, F) за шкалою: 90+ = A, 75-89 = B, 60-74 = C, 50-59 = D, <50 = F. Як переконатися, що користувач ввів саме число?

ЛЕКЦІЯ 4

Цикли: for та while

План лекції

1. Поняття циклу та цикл for: подорож світом послідовностей.

Тут ми пояснимо саму концепцію циклу – навіщо він потрібен, як він працює. Детально розберемо цикл for – найпопулярніший цикл у Python, навчимося ітерувати по різних послідовностях: рядках, списках, словниках.

2. Функція range(): генерація числових послідовностей.

Окремо зупинимося на надзвичайно корисній функції range(), яка дозволяє створювати послідовності чисел. Розберемо всі три параметри: start, stop, step, і покажемо, як за допомогою range() організовувати цикли з лічильником.

3. Цикл while: поки умова істинна.

Познайомимося з другим типом циклів – while. Розберемо, чим він відрізняється від for, у яких ситуаціях його краще використовувати, і як уникнути найстрашнішого – нескінченного циклу.

4. Керування циклом: break, continue, else та практичні прийоми.

Навчимося керувати виконанням циклів: достроково виходити (break), пропускати ітерації (continue), і розберемо цікаву особливість Python – конструкцію else у циклах. Також поговоримо про типові помилки та патерни: лічильники, акумулятори, прапорці.

1. Поняття циклу та цикл for: подорож світом послідовностей

Уявіть, що вам потрібно вивести на екран числа від 1 до 100. Без циклів вам довелося б написати 100 рядків коду: print(1), print(2), print(3) і так далі. Це було б не тільки нудно, але й абсолютно непрактично. А якщо потрібно вивести числа до 1000? До мільйона?

Саме для таких ситуацій існують **цикли**. Цикл – це конструкція, яка дозволяє виконувати певний блок коду багаторазово, поки виконується певна умова. Це одна з базових концепцій алгоритмізації, яка робить комп'ютери справді корисними – вони чудово справляються з одноманітною роботою, яку людина ненавидить.

У Python є два основних види циклів:

- **Цикл for** – використовується, коли ми знаємо, по якій послідовності потрібно пройти (наприклад, по списку студентів, по рядку тексту, по діапазону чисел).
- **Цикл while** – виконується доти, поки залишається істинною деяка умова (наприклад, "поки користувач не введе правильний пароль").

Почнемо з циклу `for`, оскільки він є найбільш "пітонічним" і використовується найчастіше.

Цикл `for`: синтаксис та призначення

Цикл `for` у Python призначений для **ітерування по послідовностях**. Він проходить по кожному елементу послідовності (рядка, списку, кортежу, словника, множини) і виконує певні дії з цим елементом.

Базовий синтаксис виглядає так:

`for` змінна `in` послідовність:

блок коду, який виконується для кожного елемента

команда¹

команда²

...

Змінна на кожній ітерації (повторенні) отримує значення чергового елемента послідовності.

Розглянемо найпростіший приклад – виведення кожного символу рядка:

```
word = "Python"
```

```
for letter in word:
```

```
    print(f"Буква: {letter}")
```

Виведе:

Буква: P

Буква: y

Буква: t

Буква: h

Буква: o

Буква: n

Зверніть увагу: ми не питали, яка довжина рядка, не створювали лічильник – Python сам пройшов по всіх символах. Це і є краса циклу `for`.

Ітерування по різних типах даних

Цикл `for` працює з будь-яким об'єктом, який є **ітерованим** (iterable). Це означає, що по ньому можна "проходити" елемент за елементом.

1. Списки (list):

```
students = ["Анна", "Богдан", "Вікторія", "Гліб"]
```

```
for student in students:  
    print(f"Студент: {student}")
```

2. Кортежі (tuple):

```
coordinates = (10, 20, 30)  
for coord in coordinates:  
    print(f"Координата: {coord}")
```

3. Словники (dict): За замовчуванням цикл for проходить по **ключах** словника.

```
person = {"name": "Олена", "age": 19, "city": "Львів"}  
for key in person:  
    print(f"Ключ: {key}, значення: {person[key]}")
```

Але є кращі способи:

```
for key, value in person.items():  
    print(f"{key}: {value}")
```

4. Множини (set):

```
unique_numbers = {1, 2, 3, 2, 1} # дублікати автоматично видаляться  
for num in unique_numbers:  
    print(f"Число: {num}") # порядок не гарантований
```

Змінна циклу: що про неї варто знати

Змінна, яку ми оголошуємо в циклі (наприклад, letter, student, num), існує тільки в межах циклу? Насправді, в Python вона залишається доступною і після завершення циклу, зберігаючи останнє присвоєне значення. Але це вважається поганим тоном – використовувати змінну циклу після його завершення.

```
for i in range(5):  
    print(i, end=" ")  
  
# після циклу  
print(f"\nОстаннє значення i: {i}") # Виведе: 4
```

Тому намагайтеся не покладатися на це, або свідомо видаляйте змінну (del i), якщо вона більше не потрібна.

Практичний приклад: обробка списку покупок

Програма для розрахунку вартості покупок

```
shopping_cart = [  
    {"item": "яблука", "price": 15, "quantity": 2},  
    {"item": "молоко", "price": 32, "quantity": 1},  
    {"item": "хліб", "price": 22, "quantity": 3},  
    {"item": "сир", "price": 120, "quantity": 0.5}  
]  
  
total = 0  
print("□ ЧЕК ПОКУПКИ")  
print("-" * 40)  
  
for product in shopping_cart:  
    item_total = product["price"] * product["quantity"]  
    total += item_total  
    print(f'{product["item"]:10} x {product["quantity"]:3} = {item_total:7.2f}  
грн')  
  
print("-" * 40)  
print(f'ЗАГАЛОМ: {total:7.2f} грн')
```

Цей приклад демонструє, як цикл for дозволяє елегантно обробити кожен елемент списку, виконати обчислення та накопичити результат.

2. Функція range(): генерація числових послідовностей

Дуже часто в програмуванні виникає потреба виконати якусь дію певну кількість разів або пройтися по числах у певному діапазоні. Для цього в Python існує чудова вбудована функція range().

Функція range() генерує послідовність цілих чисел. Вона дуже ефективна, тому що не створює одразу весь список чисел у пам'яті (про це ми поговоримо пізніше, коли вивчатимемо генератори), а "видає" числа по одному в міру необхідності.

Три форми виклику range():

1. range(stop) – генерує числа від 0 до stop-1 включно.

```
for i in range(5):
```

```
print(i, end=" ") # 0 1 2 3 4
```

Зверніть увагу: числа завжди починаються з 0 і не включають вказане число (stop). Це дуже важливо запам'ятати, щоб уникнути помилок "off-by-one".

2. range(start, stop) – генерує числа від start до stop-1 включно.

```
for i in range(2, 7):  
    print(i, end=" ") # 2 3 4 5 6
```

3. range(start, stop, step) – генерує числа від start до stop-1 з кроком step.

```
# Парні числа від 0 до 10
```

```
for i in range(0, 11, 2):  
    print(i, end=" ") # 0 2 4 6 8 10
```

```
# Зворотний порядок
```

```
for i in range(10, 0, -1):  
    print(i, end=" ") # 10 9 8 7 6 5 4 3 2 1
```

Типові використання range()

1. Повторити дію N разів:

```
for _ in range(3):  
    print("Hello!") # Виведе Hello! тричі
```

Використання _ (підкреслення) як імені змінної – це конвенція, яка показує, що значення лічильника нам не важливе.

2. Ітерувати по індексах списку:

3. Іноді нам потрібен не тільки елемент списку, але й його індекс (позиція).

```
fruits = ["яблуко", "груша", "банан", "апельсин"]  
for i in range(len(fruits)):  
    print(f"{i}: {fruits[i]}")
```

Виведе:

0: яблуко

1: груша

2: банан

3: апельсин

4. Створення числових послідовностей для обчислень:

```
# Сума чисел від 1 до 100
total = 0
for i in range(1, 101):
    total += i
print(f"Сума: {total}") # 5050
```

```
# Таблиця множення для числа 7
n = 7
for i in range(1, 11):
    print(f"{n} x {i} = {n * i}")
```

Практичний приклад: аналіз успішності студентів

Уявімо, що ми викладачі і маємо список оцінок студентів. Нам потрібно проаналізувати їх:

```
# Оцінки студентів з програмування
grades = [85, 92, 78, 90, 88, 76, 95, 89, 77, 91]

print("\n АНАЛІЗ УСПІШНОСТІ")
print("=" * 50)

# Використовуємо range для доступу за індексом
print("Детальний звіт:")
for i in range(len(grades)):
    grade = grades[i]
    if grade >= 90:
        level = "Відмінно"
    elif grade >= 75:
        level = "Добре"
    else:
        level = "Задовільно"
    print(f"Студент {i+1}: {grade} балів – {level}")
```

```
# Обчислюємо середній бал
total = 0
for grade in grades:
    total += grade
```

```

average = total / len(grades)
print(f"\nСередній бал групи: {average:.2f}")

# Знаходимо максимальну та мінімальну оцінку
max_grade = grades[0]
min_grade = grades[0]
for grade in grades:
    if grade > max_grade:
        max_grade = grade
    if grade < min_grade:
        min_grade = grade

print(f"Найвищий бал: {max_grade}")
print(f"Найнижчий бал: {min_grade}")

# Підраховуємо кількість відмінників
excellent_count = 0
for grade in grades:
    if grade >= 90:
        excellent_count += 1

print(f"Кількість відмінників: {excellent_count}")
print("=" * 50)

```

Важливе зауваження про range() у Python 3

У Python 2 функція range() повертала список. Якщо ви писали range(1000000), створювався список з мільйона елементів, що споживало багато пам'яті. У Python 3 range() повертає спеціальний об'єкт range, який генерує числа на льоту. Це дуже ефективно з точки зору пам'яті.

```

r = range(1, 1000000)
print(r)      # range(1, 1000000) – це не список!
print(len(r)) # 999999 – можна дізнатися довжину
print(r[10])  # 11 – можна отримати елемент за індексом!

```

Якщо вам дійсно потрібен список чисел, ви можете явно перетворити range у список:

```

numbers_list = list(range(1, 11))

```

```
print(numbers_list) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Поширені помилки з range()

1. Забувають, що stop не включається:

```
# Хотемо числа від 1 до 5  
for i in range(1, 5):  
    print(i) # Виведе 1,2,3,4 – а 5 немає!  
# Правильно: range(1, 6)
```

2. Плутають порядок аргументів: Завжди start, потім stop, потім step.

```
# Неправильно  
range(10, 2) # це від 10 до 2? Ні, це просто порожній діапазон, бо start >  
stop
```

```
# Правильно для зворотнього порядку  
range(10, 1, -1) # 10, 9, 8, ..., 2
```

3. Використовують range там, де він не потрібен:

```
# Погано  
fruits = ["яблуко", "груша", "банан"]  
for i in range(len(fruits)):  
    print(fruits[i])
```

```
# Добре (коли індекс не потрібен)  
for fruit in fruits:  
    print(fruit)
```

У наступному пункті ми розглянемо цикл while, який дає ще більше гнучкості, особливо коли ми не знаємо заздалегідь, скільки ітерацій потрібно виконати.

3. Цикл while: поки умова істинна

Цикл for – це чудовий інструмент, коли ми маємо справу з послідовністю: списком студентів, рядком тексту, діапазоном чисел. Але що робити, якщо ми не знаємо заздалегідь, скільки разів потрібно виконати цикл? Наприклад, ми хочемо питати у користувача пароль, поки він не введе правильний. Або ми

читаємо дані з файлу доти, поки файл не закінчиться. У таких випадках на допомогу приходить цикл `while`.

Синтаксис та логіка роботи

Цикл `while` (з англійської "поки") виконує блок коду доти, доки залишається істинною певна умова. Його синтаксис дуже простий:

`while` умова:

блок коду, який повторюється

команда¹

команда²

...

код після циклу (виконується, коли умова стає False)

Як це працює:

1. Перевіряється умова.
2. Якщо умова `True`, виконується тіло циклу.
3. Після виконання тіла повертаємося до кроку 1.
4. Якщо умова `False`, цикл завершується, і виконання продовжується з наступного після циклу рядка.

Розглянемо найпростіший приклад – лічильник:

```
counter = 1
```

```
while counter <= 5:
```

```
    print(f"Ітерація номер {counter}")
```

```
    counter += 1 # НЕ ЗАБУВАЙТЕ збільшувати лічильник!
```

```
print("Цикл завершено")
```

Виведе:

Ітерація номер 1

Ітерація номер 2

Ітерація номер 3

Ітерація номер 4

Ітерація номер 5

Цикл завершено

Зверніть увагу на рядок `counter += 1`. Це критично важливо! Якщо ми забудемо змінювати змінну, від якої залежить умова, цикл може стати нескінченним.

Нескінченні цикли та їх запобігання

Нескінченний цикл – це цикл, умова якого ніколи не стає False. У більшості випадків це помилка, яка призводить до "зависання" програми.

```
# ПОМИЛКА! Нескінченний цикл
```

```
x = 1
```

```
while x <= 5:
```

```
    print(x)
```

```
    # Забули збільшити x!
```

Ця програма буде виводити "1" вічно (точніше, поки ви не перервете її виконання комбінацією Ctrl+C). Тому завжди перевіряйте, що в тілі циклу змінюється змінна, яка впливає на умову.

Іноді нескінченні цикли створюють свідомо. Наприклад, у ігровому рушії основний цикл гри часто є нескінченним:

```
while True:
```

```
    # обробка подій
```

```
    # оновлення графіки
```

```
    # перевірка, чи не натиснув користувач кнопку "Вихід"
```

```
    # якщо так – break
```

Але в таких випадках у циклі обов'язково є умова виходу (наприклад, оператор break).

Де використовувати while, а де for?

Це важливе питання, і відповідь на нього допоможе писати більш зрозумілий код.

Ситуація	Краще використовувати
Треба пройти по всіх елементах послідовності (списку, рядку, словнику)	for
Відома кількість ітерацій	for з range()
Треба виконати дію N разів	for з range()
Кількість ітерацій заздалегідь невідома і залежить від умови	while

Ситуація	Краще використовувати
Потрібно створити нескінченний цикл	<code>while True</code>
Очікування певної події (наприклад, введення правильного пароля)	<code>while</code>

Практичні приклади використання while

1. Вгадай число (гра):

```
import random

secret = random.randint(1, 10)
guess = None
attempts = 0

print("🎮 Я загадав число від 1 до 10. Спробуй вгадати!")

while guess != secret:
    guess = int(input("Твій варіант: "))
    attempts += 1
    if guess < secret:
        print("Загадане число більше!")
    elif guess > secret:
        print("Загадане число менше!")

print(f"🎉 Вітаю! Ти вгадав число {secret} за {attempts} спроб!")
```

2. Валідація введення користувача:

```
# Просимо користувача ввести додатне число
number = -1
while number <= 0:
    number = float(input("Введіть додатне число: "))
    if number <= 0:
        print("Помилка! Число має бути додатним. Спробуйте ще раз.")
```



```
print(f'Дякую! Ви ввели {number}')
```

3. Обробка даних до певної умови:

```
# Сумуємо числа, поки не зустрінемо 0
```

```
total = 0
```

```
number = float(input("Введіть число (0 для завершення): "))
```

```
while number != 0:
```

```
    total += number
```

```
    number = float(input("Введіть наступне число (0 для завершення): "))
```

```
print(f'Сума введених чисел: {total}')
```

4. Керування циклом: break, continue, else та практичні прийоми

Цикли були б менш гнучкими, якби ми не мали можливості керувати їх виконанням зсередини. Python надає два спеціальні оператори для цього: break та continue. Крім того, у Python є унікальна особливість – конструкція else для циклів, якої немає в багатьох інших мовах програмування.

Оператор break – аварійний вихід

Оператор break негайно перериває виконання циклу, незалежно від того, чи виконана умова продовження. Програма продовжує виконання з першого рядка після циклу.

```
# Пошук першого парного числа в списку
```

```
numbers = [1, 3, 5, 7, 8, 9, 11]
```

```
for num in numbers:
```

```
    if num % 2 == 0:
```

```
        print(f'Знайдено парне число: {num}')
```

```
        break # виходимо з циклу, далі шукати не потрібно
```

```
else:
```

```
    print("Парних чисел не знайдено") # виконається, тільки якщо break не
```

```
спрацював
```

```
print("Пошук завершено")
```

break особливо корисний у циклі while, де він може бути єдиним способом виходу:

Нескінченний цикл з умовою виходу всередині

while **True**:

 command = input("Введіть команду (exit для виходу): ")

if command == "exit":

 print("До побачення!")

break

elif command == "hello":

 print("Привіт!")

else:

 print(f"Невідома команда: {command}")

Оператор continue – пропуск ітерації

Оператор continue перериває поточну ітерацію і переходить до наступної. Решта коду в тілі циклу після continue не виконується.

Вивести всі числа від 1 до 10, крім кратних 3

for i **in** range(1, 11):

if i % 3 == 0:

continue *# пропускаємо цю ітерацію*

 print(i, end=" ") *# 1 2 4 5 7 8 10*

Інший приклад – обробка даних з пропуском "поганих":

Обробка списку з пропуском від'ємних значень

data = [10, -5, 23, -1, 0, 15, -3, 7]

positive_sum = 0

positive_count = 0

for value **in** data:

if value <= 0:

continue *# пропускаємо від'ємні та нуль*

 positive_sum += value

 positive_count += 1

print(f"Додатних чисел: {positive_count}, їх сума: {positive_sum}")

Конструкція else у циклах – унікальна особливість Python

У Python цикли можуть мати блок else, який виконується **тільки якщо** цикл завершився **нормально** (без break). Це спочатку може здаватися неінтуїтивним, але на практиці дуже корисно.

Пошук простого числа

```
n = int(input("Введіть число: "))
```

```
for i in range(2, n):
```

```
    if n % i == 0:
```

```
        print(f"{n} не є простим (ділиться на {i})")
```

```
        break
```

```
else:
```

Цей блок виконається, тільки якщо break НЕ спрацював

```
    print(f"{n} є простим числом!")
```

Без else нам довелося б використовувати додаткову змінну-прапорець:

Так довелося б писати без else

```
n = int(input("Введіть число: "))
```

```
is_prime = True
```

```
for i in range(2, n):
```

```
    if n % i == 0:
```

```
        print(f"{n} не є простим (ділиться на {i})")
```

```
        is_prime = False
```

```
        break
```

```
if is_prime:
```

```
    print(f"{n} є простим числом!")
```

Як бачите, else робить код чистішим і зрозумілішим.

Лічильники та акумулятори – класичні прийоми роботи з циклами

У циклах часто використовують дві важливі концепції:

1. **Лічильник (counter)** – змінна, яка рахує кількість ітерацій або кількість елементів, що задовольняють умову.

2. **Акумулятор (accumulator)** – змінна, яка накопичує результат (суму, добуток тощо).

Приклад з лічильником та акумулятором

```
numbers = [10, 23, 5, 17, 8, 12, 4, 19]
```

```
count_even = 0    # лічильник
```

```
sum_even = 0      # акумулятор
```

```
for num in numbers:
```

```
    if num % 2 == 0:
```

```
        count_even += 1
```

```
        sum_even += num
```

```
print(f'Парних чисел: {count_even}')
```

```
print(f'Сума парних чисел: {sum_even}')
```

```
print(f'Середнє парних: {sum_even / count_even if count_even > 0 else 0}')
```

Прапорці (flags) – ще один корисний прийом:

Перевірка, чи всі числа в списку додатні

```
numbers = [10, 23, -5, 17, 8]
```

```
all_positive = True # прапорець
```

```
for num in numbers:
```

```
    if num <= 0:
```

```
        all_positive = False
```

```
        break
```

```
if all_positive:
```

```
    print("Всі числа додатні")
```

```
else:
```

```
    print("Є від'ємні числа або нуль")
```

Типові помилки при роботі з циклами

1. Нескінченний цикл (забули змінити лічильник) – вже розглядали.
2. Помилка "off-by-one" – неправильне визначення меж.

Хочемо вивести числа від 1 до 5

```
i = 1
```

```
while i < 5: # ПОМИЛКА! Виведе 1,2,3,4 (пропустили 5)
```

```
    print(i)
```

```
i += 1
```

Правильно: while i <= 5

3. Зміна списку під час ітерації – небезпечна практика.

ПОГАНО! Не видаляйте елементи списку під час ітерації

```
numbers = [1, 2, 3, 4, 5]
```

```
for num in numbers:
```

```
    if num % 2 == 0:
```

```
        numbers.remove(num)    # Це призведе до непередбачуваних
```

результатів!

Краще створити новий список

```
numbers = [num for num in numbers if num % 2 != 0]
```

Практичний приклад: програма для управління завданнями

Об'єднаємо всі вивчені концепції в невелику, але корисну програму:

TODO-менеджер (спрацює в терміналі)

```
print("📅 МІЙ ТОДО-ЛИСТ")
```

```
print("=" * 50)
```

```
tasks = []
```

```
while True:
```

```
    print("\nКоманди:")
```

```
    print(" add - додати завдання")
```

```
    print(" list - показати всі завдання")
```

```
    print(" done - позначити завдання як виконане")
```

```
    print(" exit - вийти")
```

```
command = input("Введіть команду: ").strip().lower()
```

```
if command == "exit":
```

```
    print("До побачення!")
```

```
    break
```

```
elif command == "add":
```

```
    task = input("Опишіть завдання: ")
```

```
    tasks.append({"task": task, "done": False})
```

```

print(f"✓ Завдання додано!")

elif command == "list":
    if not tasks: # перевірка на порожній список
        print("Список завдань порожній")
        continue

    print("\nВаші завдання:")
    for i, task in enumerate(tasks, 1):
        status = "✓" if task["done"] else " "
        print(f"{i}. [{status}] {task['task']}")

elif command == "done":
    if not tasks:
        print("Немає завдань для позначення")
        continue

    # Показуємо тільки невиконані завдання
    pending = [(i, task) for i, task in enumerate(tasks) if not task["done"]]

    if not pending:
        print("Всі завдання вже виконані!")
        continue

    print("Невиконані завдання:")
    for idx, (original_idx, task) in enumerate(pending, 1):
        print(f"{idx}. {task['task']}")

    try:
        choice = int(input("Яке завдання виконано (номер)? ")) - 1
        if 0 <= choice < len(pending):
            task_idx = pending[choice][0]
            tasks[task_idx]["done"] = True
            print("✓ Завдання позначено як виконане!")
        else:
            print("✗ Невірний номер")
    except ValueError:

```

```
print("✗ Введіть число")
```

```
else:
```

```
print("✗ Невідома команда")
```

```
print("=" * 50)
```

Ця програма демонструє:

- Використання while True з умовою виходу через break
- Перевірку на порожні списки (if not tasks)
- Роботу з лічильниками та індексами
- Обробку винятків (поки що поверхнево)
- Різні команди для взаємодії з користувачем

У наступній лекції ми розглянемо вкладені цикли – потужний інструмент для роботи з двовимірними структурами даних та створення складніших алгоритмів.

Питання для обговорення

1. Поясніть різницю між циклами for та while. У яких ситуаціях краще використовувати for, а в яких – while?
2. Як працює функція range()? Напишіть приклади використання range з одним, двома та трьома аргументами. Що буде результатом list(range(5, 1, -1))?
3. Проаналізуйте код. Скільки разів виконається цикл? Чому?

```
for i in range(5):
```

```
    print(i)
```

```
    i = 10
```

4. Що таке нескінченний цикл? Напишіть приклад нескінченного циклу while. Як його можна перервати? Як уникнути випадкового створення нескінченного циклу?
5. Поясніть різницю між операторами break та continue. Наведіть приклад, де використання break є виправданим, і приклад, де continue робить код ефективнішим.
6. Для чого в циклах використовується конструкція else? У якому випадку блок else виконується, а в якому – ні? Наведіть практичний приклад, де else робить код зрозумілішим.

7. Напишіть програму, яка за допомогою циклу `while` запитує у користувача пароль доти, доки він не введе правильний. Використовуйте `break` для виходу з циклу.

8. Що таке лічильник та акумулятор у контексті циклів? Напишіть програму, яка за допомогою циклу `for` обчислює суму всіх чисел від 1 до 100 та їх кількість.

9. Чому код `for i in range(len(my_list))`: іноді вважають менш "пітонічним", ніж `for item in my_list`? Коли все ж таки потрібно використовувати перший варіант?

10. Проаналізуйте код. Поясніть, що він робить і чому результат може бути неочікуваним.

```
text = "Hello"  
for char in text:  
    text += char  
print(text)
```


ЛЕКЦІЯ 5

Вкладені цикли та складні алгоритми

План лекції

1. Концепція вкладених циклів: цикл у циклі.

Пояснимо саму ідею вкладених структур – як один цикл працює всередині іншого, як змінюються лічильники, чому загальна кількість ітерацій дорівнює добутку. Розберемо найпростіші приклади: таблиця множення, виведення прямокутника з зірочок.

2. Обробка двовимірних структур даних: матриці та таблиці.

Покажемо, як вкладені цикли природно відповідають структурі матриць (списків списків). Навчимося створювати, заповнювати та обробляти двовимірні дані: суми елементів, пошук максимумів, транспонування матриць.

3. Генерація патернів та практичне застосування.

Розглянемо, як за допомогою вкладених циклів малювати різноманітні фігури: трикутники, ромби, ялинки. Це не тільки весело, але й чудово тренує алгоритмічне мислення. Також поговоримо про комбінування циклів з умовами.

4. Ефективність та альтернативи: $O(n^2)$ та list comprehensions.

Обговоримо важливе поняття складності алгоритмів – квадратичну залежність $O(n^2)$, коли вкладені цикли можуть бути проблемою. Розглянемо типові помилки та способи оптимізації. Наприкінці познайомимося з більш елегантною альтернативою – списковими включеннями (list comprehensions) для створення двовимірних структур.

1. Концепція вкладених циклів: цикл у циклі

Досі ми працювали з циклами, які виконували одну послідовність дій. Але в реальному житті часто виникають задачі, де потрібно виконати повторювані дії всередині інших повторюваних дій. Уявіть, що ви хочете роздрукувати розклад занять для кожного дня тижня. Для кожного дня (зовнішній цикл) вам потрібно вивести список пар (внутрішній цикл). Саме тут нам знадобляться **вкладені цикли**.

Що таке вкладені цикли?

Вкладений цикл – це цикл, який знаходиться всередині тіла іншого циклу. Для кожної ітерації зовнішнього циклу внутрішній цикл виконується повністю.

Давайте розглянемо найпростіший приклад – виведення координат точок на сітці:

```
print("Координати точок (x, y):")
```

```

for x in range(3):      # зовнішній цикл (x від 0 до 2)
    for y in range(2):  # внутрішній цикл (y від 0 до 1)
        print(f"({x}, {y})", end=" ")
    print() # перехід на новий рядок після кожного x

```

Виведе:

```

(0, 0) (0, 1)
(1, 0) (1, 1)
(2, 0) (2, 1)

```

Зверніть увагу, як це працює:

1. x = 0 (перша ітерація зовнішнього циклу)
 - Внутрішній цикл виконується повністю: y = 0, потім y = 1
2. x = 1 (друга ітерація зовнішнього циклу)
 - Внутрішній цикл знову виконується повністю: y = 0, y = 1
3. x = 2 (третя ітерація зовнішнього циклу)
 - Внутрішній цикл виконується повністю: y = 0, y = 1

Загальна кількість ітерацій внутрішнього циклу дорівнює добутку кількості ітерацій зовнішнього на кількість ітерацій внутрішнього. У нашому прикладі: $3 \times 2 = 6$ точок.

Таблиця множення – класичний приклад

Найвідоміший приклад використання вкладених циклів – це генерація таблиці множення:

```

print("ТАБЛИЦЯ МНОЖЕННЯ")
print("=" * 50)

for i in range(1, 10): # зовнішній цикл – множник i
    for j in range(1, 10): # внутрішній цикл – множник j
        product = i * j
        print(f"{i} × {j} = {product:2}", end=" ")
    print() # перехід на новий рядок після кожного i
print("=" * 50)

```

Цей код створить класичну таблицю множення 9×9. Зверніть увагу на форматування: {product:2} означає, що число займає щонайменше 2 позиції, що робить таблицю акуратною.

Малювання прямокутників за допомогою зірочок

Ще один чудовий приклад для розуміння вкладених циклів – малювання прямокутників з символів:

Малюємо прямокутник 5×10 із зірочок

height = 5

width = 10

```
for i in range(height):      # зовнішній цикл – рядки
    for j in range(width):    # внутрішній цикл – стовпчики
        print("*", end="")
    print() # перехід на новий рядок
```

Виведе:

```
*****
*****
*****
*****
*****
```

Зверніть увагу: внутрішній цикл малює один рядок зірочок, а зовнішній повторює це height разів, створюючи потрібну висоту.

Як працюють лічильники у вкладених циклах

Важливо розуміти, як змінюються лічильники. Даваймо додамо виведення індексів:

```
for i in range(3):
    print(f"Початок ітерації i = {i}")
    for j in range(2):
        print(f" всередині: j = {j}")
    print(f"Кінець ітерації i = {i}")
    print("-" * 20)
```

Ви побачите, що для кожного значення i внутрішній цикл проходить всі значення j від початку до кінця.

Практичний приклад: розклад занять

Уявімо, що ми створюємо програму для виведення розкладу:

Розклад занять

```
days = ["Понеділок", "Вівторок", "Середа", "Четвер", "П'ятниця"]
lessons = ["Математика", "Фізика", "Програмування", "Англійська"]
```

```
print("📅 РОЗКЛАД ЗАНЯТЬ")
print("=" * 50)
```

```
for day in days:
    print(f"\n{day}:")
    for i, lesson in enumerate(lessons, 1):
        print(f" {i}-й урок: {lesson}")
```

```
print("=" * 50)
```

2. Обробка двовимірних структур даних: матриці та таблиці

Вкладені цикли стають особливо корисними, коли ми працюємо з двовимірними структурами даних – матрицями, таблицями, зображеннями (як піксельні сітки). У Python ми можемо представити матрицю як список списків.

Створення матриць (списків списків)

Створюємо матрицю 3×3

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

```
print("Матриця:")
for row in matrix:
    for element in row:
        print(element, end=" ")
    print() # новий рядок після кожного рядка матриці
```

Виведе:

```
1 2 3
4 5 6
7 8 9
```

Зверніть увагу: зовнішній цикл проходить по **рядках** матриці, внутрішній – по **елементах** всередині кожного рядка.

Доступ за індексами

Часто нам потрібен не тільки сам елемент, але й його позиція (індекс рядка та стовпчика):

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print("Елементи матриці з індексами:")
for i in range(len(matrix)):      # i – індекс рядка
    for j in range(len(matrix[i])): # j – індекс стовпчика
        print(f"matrix[{i}][{j}] = {matrix[i][j]}")
```

Обчислення суми всіх елементів матриці

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

total = 0
for i in range(len(matrix)):
    for j in range(len(matrix[i])):
        total += matrix[i][j]

print(f"Сума всіх елементів: {total}") # 45
```

Пошук максимального елемента

```
matrix = [
```

```
[1, 2, 3],  
[4, 5, 6],  
[7, 8, 9]  
]
```

```
max_element = matrix[0][0]  
max_row = 0  
max_col = 0
```

```
for i in range(len(matrix)):  
    for j in range(len(matrix[i])):  
        if matrix[i][j] > max_element:  
            max_element = matrix[i][j]  
            max_row = i  
            max_col = j
```

```
print(f"Максимальний елемент: {max_element}")  
print(f"Знаходиться у рядку {max_row}, стовпчику {max_col}")
```

Створення матриці заданого розміру

Часто потрібно створити матрицю певного розміру і заповнити її, наприклад, нулями:

```
# Створюємо матрицю 4×5, заповнену нулями  
rows = 4  
cols = 5  
matrix = []  
  
for i in range(rows):  
    row = [] # створюємо новий рядок  
    for j in range(cols):  
        row.append(0) # заповнюємо нулями  
    matrix.append(row) # додаємо рядок до матриці  
  
print("Матриця 4×5 з нулів:")  
for row in matrix:  
    for element in row:  
        print(element, end=" ")
```

```
print()
```

Але в Python є більш елегантний спосіб (хоча з ним треба бути обережним):

```
# Так працює, але є нюанс!
```

```
rows, cols = 4, 5
```

```
matrix = [[0] * cols for _ in range(rows)] # спискове включення (list comprehension)
```

Про спискові включення ми детальніше поговоримо в пункті 4, але поки запам'ятайте цей шаблон.

Транспонування матриці

Транспонування – це коли рядки стають стовпчиками, а стовпчики – рядками:

```
# Вхідна матриця
```

```
matrix = [  
    [1, 2, 3],  
    [4, 5, 6]  
]
```

```
# Вона має розмір 2×3
```

```
print("Вхідна матриця:")
```

```
for row in matrix:
```

```
    print(row)
```

```
# Транспонована матриця буде розміром 3×2
```

```
rows = len(matrix)
```

```
cols = len(matrix[0])
```

```
# Створюємо матрицю для результату
```

```
transposed = [[0 for _ in range(rows)] for _ in range(cols)]
```

```
for i in range(rows):
```

```
    for j in range(cols):
```

```
        transposed[j][i] = matrix[i][j]
```

```
print("\nТранспонована матриця:")
```

```
for row in transposed:
    print(row)
Виведе:
```

Вхідна матриця:

```
[1, 2, 3]
[4, 5, 6]
```

Транспонована матриця:

```
[1, 4]
[2, 5]
[3, 6]
```

Практичний приклад: обробка оцінок студентів

Уявімо, що ми маємо таблицю оцінок студентів з різних предметів:

```
# Таблиця успішності: рядки – студенти, стовпчики – предмети
# [студент1, студент2, ...]
grades = [
    [85, 90, 78, 92], # Студент 1: математика, фізика, програмування,
англійська
    [75, 88, 95, 70], # Студент 2
    [95, 92, 88, 96], # Студент 3
    [60, 75, 80, 65], # Студент 4
    [100, 98, 95, 99] # Студент 5
]
```

```
subjects = ["Математика", "Фізика", "Програмування", "Англійська"]
```

```
print("██ АНАЛІЗ УСПІШНОСТІ ГРУПИ")
print("=" * 60)
```

1. Середній бал кожного студента

```
print("\n█ СЕРЕДНІ БАЛИ СТУДЕНТІВ:")
for i in range(len(grades)):
    student_sum = 0
    for grade in grades[i]:
```



```

    student_sum += grade
    student_avg = student_sum / len(grades[i])
    print(f'Студент {i+1}: {student_avg:.2f}')

```

2. Середній бал з кожного предмету

```

print("\n📁 СЕРЕДНІ БАЛИ З ПРЕДМЕТІВ:")
for j in range(len(subjects)):
    subject_sum = 0
    for i in range(len(grades)):
        subject_sum += grades[i][j]
    subject_avg = subject_sum / len(grades)
    print(f'{subjects[j]}: {subject_avg:.2f}')

```

3. Знаходимо найкращого студента (за сумою балів)

```

best_student = 0
best_sum = 0
for i in range(len(grades)):
    current_sum = sum(grades[i]) # можна використати вбудовану функцію
sum
    if current_sum > best_sum:
        best_sum = current_sum
        best_student = i + 1

print(f"\n🏆 Найкращий студент: Студент {best_student} (сума балів:
{best_sum})")

```

4. Знаходимо предмет, з яким найбільше проблем (найнижчий середній)

```

min_avg = float('inf')
worst_subject = ""
for j in range(len(subjects)):
    subject_sum = 0
    for i in range(len(grades)):
        subject_sum += grades[i][j]
    subject_avg = subject_sum / len(grades)
    if subject_avg < min_avg:
        min_avg = subject_avg
        worst_subject = subjects[j]

```

```
print(f"📉 Найскладніший предмет: {worst_subject} (середній бал: {min_avg:.2f})")
print("=" * 60)
```

Робота з "зубчастими" матрицями

У Python списки списків не обов'язково мають бути прямокутними – кожен "рядок" може мати свою довжину:

Зубчаста матриця (рядки різної довжини)

```
jagged = [
    [1, 2, 3],
    [4, 5],
    [6, 7, 8, 9],
    [10]
]
```

```
print("Зубчаста матриця:")
for i, row in enumerate(jagged):
    print(f"Рядок {i}: {row}, довжина: {len(row)}")
```

Обробка такої матриці вимагає обережності

```
total_sum = 0
for row in jagged:
    for element in row:
        total_sum += element
print(f"Сума всіх елементів: {total_sum}")
```

У наступних пунктах ми розглянемо, як за допомогою вкладених циклів створювати красиві патерни, а також поговоримо про ефективність та альтернативні підходи.

3. Генерація патернів та практичне застосування

Вкладені цикли відкривають безмежні можливості для творчості. За їх допомогою можна створювати різноманітні візерунки, геометричні фігури та

навіть елементи графіки в текстових додатках. Це не тільки цікаво, але й надзвичайно корисно для тренування алгоритмічного мислення.

Малювання прямокутників та квадратів

Почнемо з найпростішого – заповненого прямокутника. Ми вже робили це, але тепер ускладнимо задачу:

```
# Заповнений прямокутник
```

```
width = 10
```

```
height = 5
```

```
symbol = "*"
```

```
print("ЗАПОВНЕНИЙ ПРЯМОКУТНИК")
```

```
for i in range(height):
```

```
    for j in range(width):
```

```
        print(symbol, end="")
```

```
    print()
```

А що, якщо нам потрібен лише контур прямокутника?

```
# Порожнистий прямокутник (тільки рамка)
```

```
width = 10
```

```
height = 5
```

```
symbol = "*"
```

```
print("\nПОРОЖНИСТИЙ ПРЯМОКУТНИК")
```

```
for i in range(height):
```

```
    for j in range(width):
```

```
        # Друкуємо символ, якщо це перший або останній рядок,
```

```
        # або перший або останній стовпчик
```

```
        if i == 0 or i == height - 1 or j == 0 or j == width - 1:
```

```
            print(symbol, end="")
```

```
        else:
```

```
            print(" ", end="")
```

```
    print()
```

Трикутники – класика алгоритмічних задач

Прямокутні трикутники бувають різних орієнтацій. Почнемо з найпростішого – лівий нижній кут:

```
# Трикутник (лівий нижній кут)
size = 7
print("ТРИКУТНИК 1 (лівий нижній кут):")
for i in range(size):
    for j in range(i + 1):
        print("*", end="")
    print()
```

Виведе:

```
*
**
***
****
*****
*****
*****
*****
```

А тепер перевернемо його – лівий верхній кут:

```
# Трикутник (лівий верхній кут)
size = 7
print("\nТРИКУТНИК 2 (лівий верхній кут):")
for i in range(size):
    for j in range(size - i):
        print("*", end="")
    print()
```

Для трикутників з правим вирівнюванням нам знадобляться пробіли:

```
# Трикутник (правий нижній кут)
size = 7
print("\nТРИКУТНИК 3 (правий нижній кут):")
for i in range(size):
    # Друкуємо пробіли
    for j in range(size - i - 1):
        print(" ", end="")
    # Друкуємо зірочки
    for j in range(i + 1):
```

```
print("*", end="")
print()
```

Ромб – симетрична фігура

Створення ромба вимагає більш складної логіки:

```
# Ромб
size = 5
print("\nPOMБ:")

# Верхня половина (включно з серединою)
for i in range(size):
    # Пробіли зменшуються
    for j in range(size - i - 1):
        print(" ", end="")
    # Зірочки збільшуються (непарна кількість)
    for j in range(2 * i + 1):
        print("*", end="")
    print()

# Нижня половина
for i in range(size - 2, -1, -1):
    # Пробіли збільшуються
    for j in range(size - i - 1):
        print(" ", end="")
    # Зірочки зменшуються
    for j in range(2 * i + 1):
        print("*", end="")
    print()
```

Цей код створить симетричний ромб. Зверніть увагу на використання негативного кроку в range() для проходження знизу вгору.

Ялинка – святковий настрій

Об'єднаємо трикутники, щоб створити ялинку:

```
# Новорічна ялинка
levels = 4
height_per_level = 5
```

```
print("\n🌲 НОВОРІЧНА ЯЛИНКА:")
```

```
for level in range(levels):
    # Верхівка кожного рівня
    for i in range(height_per_level):
        # Пробіли для центрування (зменшуються з кожним рівнем)
        for j in range(levels * height_per_level - i - level):
            print(" ", end="")
        # Зірочки
        for j in range(2 * (i + level) + 1):
            print("*", end="")
        print()

# Стовбур
for i in range(3):
    for j in range(levels * height_per_level - 2):
        print(" ", end="")
    print("###")
```

Комбінування циклів з умовними операторами

Дуже часто вкладений цикл містить умови, які змінюють поведінку залежно від позиції. Класичний приклад – гра "Життя" (шаблон для малювання шахової дошки):

```
# Шахова дошка
size = 8
print("\n♔ ШАХОВА ДОШКА:")

for i in range(size):
    for j in range(size):
        # Парність суми індексів визначає колір клітинки
        if (i + j) % 2 == 0:
            print("■", end=" ") # біла клітинка
        else:
            print("□", end=" ") # чорна клітинка
    print()
```

Практичний приклад: генерація таблиці з кольоровим виділенням

Уявімо, що ми створюємо програму для візуалізації температури:

```
# Теплова карта (текстова версія)
import random

# Генеруємо випадкові температури
temperatures = []
for i in range(7): # дні тижня
    day_temps = []
    for j in range(24): # години
        day_temps.append(random.randint(15, 30))
    temperatures.append(day_temps)

days = ["Пн", "Вт", "Ср", "Чт", "Пт", "Сб", "Нд"]
hours = list(range(24))

print("ТЕПЛОВА КАРТА ТЕМПЕРАТУР")
print("  " + " ".join([f"{h:2}" for h in hours]))

for i, day in enumerate(days):
    print(f"{day}: ", end="")
    for temp in temperatures[i]:
        if temp < 18:
            symbol = "❄" # холодно
        elif temp < 22:
            symbol = "🌡" # нормально
        elif temp < 26:
            symbol = "☀" # тепло
        else:
            symbol = "🔥" # жарко
    print(f"{symbol} ", end="")
print()
```

4. Ефективність та альтернативи: $O(n^2)$ та list comprehensions

Коли ми пишемо вкладені цикли, особливо для обробки великих обсягів даних, важливо розуміти, наскільки ефективним є наш код. Тут ми вперше стикаємося з поняттям **складності алгоритмів**.

Поняття про $O(n^2)$ – квадратична складність

Якщо у нас є один цикл, який виконується n разів, то кількість операцій пропорційна n . Це лінійна складність – $O(n)$. Але коли ми вкладаємо один цикл в інший, і обидва виконуються n разів, загальна кількість операцій стає $n \times n = n^2$. Це називається **квадратичною складністю** і позначається як $O(n^2)$.

$O(n)$ – лінійна складність

$n = 1000$

```
for i in range(n):
```

```
    print(i) # 1000 операцій
```

$O(n^2)$ – квадратична складність

```
for i in range(n):
```

```
    for j in range(n):
```

```
        print(i, j) # 1 000 000 операцій!
```

Чому це важливо? Тому що зростання кількості даних призводить до різкого збільшення часу виконання:

n	$O(n)$ операцій	$O(n^2)$ операцій
10	10	100
100	100	10 000
1000	1000	1 000 000
10 000	10 000	100 000 000

Як бачите, при $n = 10\,000$ різниця стає драматичною: 10 тисяч операцій проти 100 мільйонів!

Коли вкладені цикли є необхідністю?

Деякі задачі за своєю природою вимагають квадратичної складності:

- Порівняння кожного елемента з кожним у двох списках
- Обробка матриць та двовимірних масивів
- Деякі алгоритми сортування (бульбашка, вставками)

Але часто ми можемо оптимізувати код, зменшивши кількість ітерацій.

Оптимізація вкладених циклів

1. Зменшення кількості ітерацій внутрішнього циклу:

Погано: зайві ітерації

```
for i in range(n):
    for j in range(n):
        if j <= i: # половина ітерацій марна
            # робимо щось
            pass
```

Краще: внутрішній цикл тільки до i

```
for i in range(n):
    for j in range(i + 1):
        # робимо щось
        pass
```

2. Винесення інваріантних обчислень:

Погано: одне й те саме обчислюється багато разів

```
for i in range(n):
    for j in range(n):
        result = i * j * (some_complex_value / 100) # some_complex_value не змінюється
        print(result)
```

Краще: обчислюємо один раз

```
factor = some_complex_value / 100
for i in range(n):
    for j in range(n):
        result = i * j * factor
        print(result)
```

3. Ранній вихід (break):

Пошук пари чисел, що в сумі дають target

```
numbers = [1, 2, 3, 4, 5]
target = 7
found = False
```

```

for i in range(len(numbers)):
    for j in range(i + 1, len(numbers)):
        if numbers[i] + numbers[j] == target:
            print(f"Знайдено: {numbers[i]} + {numbers[j]} = {target}")
            found = True
            break # вихід з внутрішнього циклу
if found:
    break # вихід із зовнішнього циклу

```

List Comprehensions – елегантна альтернатива

Python пропонує більш компактний та часто швидший спосіб створення списків – **спискові включення (list comprehensions)**. Вони особливо корисні для створення двовимірних структур.

Синтаксис list comprehension:

```
new_list = [вираз for елемент in послідовність if умова]
```

Приклади:

Створення списку квадратів чисел

```

squares = [x**2 for x in range(10)]
print(squares) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

```

З умовою: тільки парні числа

```

even_squares = [x**2 for x in range(20) if x % 2 == 0]
print(even_squares) # [0, 4, 16, 36, 64, 100, 144, 196, 256, 324]

```

List comprehensions для двовимірних структур

Створення матриці 5×5 з нулями

```

matrix = [[0 for _ in range(5)] for _ in range(5)]
print(matrix)

```

Створення одиничної матриці

```

identity = [[1 if i == j else 0 for j in range(5)] for i in range(5)]
for row in identity:
    print(row)

```

Порівняння продуктивності

List comprehensions не тільки компактніші, але й часто працюють швидше за звичайні цикли:

```
import time

n = 1000

# Звичайний вкладений цикл
start = time.time()
result1 = []
for i in range(n):
    row = []
    for j in range(n):
        row.append(i * j)
    result1.append(row)
end = time.time()
print(f"Цикли: {end - start:.4f} сек")
```

```
# List comprehension
start = time.time()
result2 = [[i * j for j in range(n)] for i in range(n)]
end = time.time()
print(f"List comprehension: {end - start:.4f} сек")
```

Зазвичай list comprehension працює в 1.5-2 рази швидше.

Типові помилки у вкладених структурах

1. **Плутанина з індексами** – особливо при роботі з матрицями.
2. **Некоректне копіювання** – створення посилань замість копій.

```
# ПОМИЛКА! Всі рядки будуть посилатися на той самий список
matrix = [[0] * 5] * 5
matrix[0][0] = 1
print(matrix) # [[1, 0, 0, 0, 0], [1, 0, 0, 0, 0], ...]
```

```
# ПРАВИЛЬНО
matrix = [[0] * 5 for _ in range(5)]
```

3. **Забування про змінні циклу** – використання однієї змінної для зовнішнього і внутрішнього циклу.

Практичний приклад: гра "Сапер" (спрощена версія)

Об'єднаємо всі знання для створення спрощеної версії поля для гри "Сапер":

```
import random

# Розміри поля
rows, cols = 8, 8
mines_count = 10

# Створюємо порожнє поле
field = [[0 for _ in range(cols)] for _ in range(rows)]

# Розставляємо міни
mines_placed = 0
while mines_placed < mines_count:
    r = random.randint(0, rows - 1)
    c = random.randint(0, cols - 1)
    if field[r][c] != -1: # якщо тут ще немає міни
        field[r][c] = -1
        mines_placed += 1

# Обчислюємо числа навколо мін
for i in range(rows):
    for j in range(cols):
        if field[i][j] == -1: # якщо це міна, пропускаємо
            continue

    # Перевіряємо всі сусідні клітинки
    mines_nearby = 0
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if di == 0 and dj == 0:
                continue
            ni, nj = i + di, j + dj
            if 0 <= ni < rows and 0 <= nj < cols and field[ni][nj] == -1:
                mines_nearby += 1
```

```
field[i][j] = mines_nearby
```

```
# Виводимо поле (красиво)
```

```
print("СПРОЩЕНИЙ САПЕР")
```

```
print(" " + " ".join([f"{c:2}" for c in range(cols)]))
```

```
print(" " + "-" * (cols * 3))
```

```
for i in range(rows):
```

```
    print(f"{i:2}|" , end=" ")
```

```
    for j in range(cols):
```

```
        if field[i][j] == -1:
```

```
            print("🍌", end=" ")
```

```
        else:
```

```
            print(f"{field[i][j]:2}", end=" ")
```

```
    print()
```

Цей приклад демонструє:

- Потрійні вкладені цикли (для кожного елемента перевіряємо всіх сусідів)
- Роботу з границями матриці (перевірка $0 \leq i < \text{rows}$)
- Умовну логіку всередині циклів
- Створення та заповнення двовимірних структур

У наступній лекції ми перейдемо до детальної роботи з рядками – ще однією фундаментальною темою, де вкладені цикли також часто знаходять застосування.

Питання для обговорення

1. Поясніть, як працюють вкладені цикли. Скільки разів виконається тіло внутрішнього циклу, якщо зовнішній виконується m разів, а внутрішній – n разів?
2. Напишіть програму з використанням вкладених циклів для виведення таблиці множення від 1 до 5.
3. Як за допомогою вкладених циклів створити наступний патерн?

```
*
```

```
**
```

4. Що таке матриця в програмуванні? Як представити матрицю в Python? Напишіть код для створення матриці 3×3 , заповненої нулями.

5. Напишіть функцію, яка приймає матрицю (список списків) і повертає суму всіх її елементів, використовуючи вкладені цикли.

6. Поясніть поняття "квадратична складність" $O(n^2)$. Чому алгоритми з двома вкладеними циклами можуть бути повільними для великих обсягів даних? Наведіть приклад, коли зі збільшенням вхідних даних у 10 разів час виконання зростає у 100 разів.

7. Як можна оптимізувати вкладені цикли? Запропонуйте хоча б дві стратегії зменшення кількості ітерацій.

8. Що таке list comprehensions? Перепишіть наступний код з використанням list comprehension для створення списку квадратів чисел від 1 до 10.

```
squares = []  
for i in range(1, 11):  
    squares.append(i**2)
```

9. Яка помилка в наступному коді, що намагається створити матрицю 3×3 ? Чому вона виникає?

```
matrix = [[0] * 3] * 3  
matrix[0][0] = 1  
print(matrix)
```

10. Напишіть програму, яка за допомогою вкладених циклів створює "теплову карту" у вигляді текстової таблиці, де кожна клітинка заповнена символом залежно від значення (наприклад, * для високих значень, . для середніх, пробіл для низьких). Як би ви змінили цю програму, щоб вона працювала з реальними даними (наприклад, температурами за місяцями)?

ЛЕКЦІЯ 6

Робота з рядками

План лекції

1. Рядки як послідовності: створення, індексація та незмінність.

Почнемо з основ: як створювати рядки, чому потрібні різні види лапок, що таке екранування. Пояснимо концепцію рядка як послідовності символів, розберемо індексацію (позитивну та негативну) і, найголовніше, фундаментальну властивість рядків – незмінність (immutability), яка часто викликає подив у початківців.

2. Зрізи (slicing): мистецтво виділення підрядків.

Детально розберемо один з найпотужніших інструментів Python – зрізи. Навчимося отримувати підрядки, перевертати рядки, брати кожен другий символ. Покажемо, як пропущені індекси роблять синтаксис елегантним та зручним.

3. Основні методи рядків: трансформація, пошук та перевірка.

Познайомимося з багатою стандартною бібліотекою методів рядків: зміна регістру, очищення від пробілів, пошук підрядків, заміна, перевірка вмісту. Розберемо, які методи повертають нові рядки, а які – булеві значення чи індекси.

4. Розбиття, з'єднання та форматування рядків.

Навчимося перетворювати рядки на списки і навпаки – це критично важливо для обробки даних. Поглибимо знання про форматування: від старих методів до сучасних f-strings, які роблять код чистим і зрозумілим.

1. Рядки як послідовності: створення, індексація та незмінність

Рядки (strings) – це один з найважливіших типів даних у програмуванні. Ми використовуємо їх для роботи з текстом: імена користувачів, повідомлення, адреси електронної пошти, вміст файлів – усе це рядки. У Python рядки реалізовані дуже потужно і водночас елегантно.

Створення рядків: лапки мають значення

У Python є кілька способів створити рядок, і вибір залежить від ситуації:

Одинарні лапки – найпоширеніший спосіб

```
name = 'Олена'
```

```
message = 'Привіт, світе!'
```

Подвійні лапки – теж дуже поширені

```
name = "Олена"
```

```
message = "Привіт, світе!"
```

Потрійні лапки – для багаторядкових рядків

```
poem = """
```

```
У лузі червона калина
```

```
Похилилася,
```

```
Чогось наша славна Україна
```

```
Засмутилася.
```

```
"""
```

```
print(poem)
```

Потрійні лапки зберігають всі перенесення рядків та пробіли

Чому потрібні різні лапки?

Головна причина – можливість використовувати лапки всередині рядка без екранування:

Якщо рядок містить одинарні лапки, зручно використати подвійні

```
sentence = "Він сказав: 'Привіт!'"
```

Якщо рядок містить подвійні лапки, зручно використати одинарні

```
sentence = 'Він сказав: "Привіт!"'
```

Якщо є й ті, й ті – доведеться екранувати

```
sentence = "Він сказав: \"Привіт!\" і додав: 'Як справи?'"
```

Екранування спеціальних символів

Для вставки спеціальних символів використовується зворотній слеш \:

\n – новий рядок

```
print("Перший рядок\nДругий рядок")
```

\t – табуляція

```
print("Ім'я:\tОлена")
```

\\ – сам символ зворотнього слеша

```
print("Шлях до файлу: C:\\Users\\Name\\Desktop")
```



```
# \' i \' – лапки всередині рядка  
print("Він сказав: \"Python – це круто!\")
```

Рядки як послідовності

У Python рядок – це **послідовність символів**. Це означає, що з рядком можна працювати так само, як зі списком: звертатися до окремих символів за індексом, отримувати довжину, ітерувати по символах.

```
word = "Python"  
print(len(word)) # 6 – довжина рядка
```

```
# Ітерування по символах  
for char in word:  
    print(char, end=" ") # P y t h o n
```

Індексація: позитивна та негативна

Кожен символ у рядку має свій індекс – позицію. В Python індекси починаються з 0.

Рядок: P y t h o n
Індекс: 0 1 2 3 4 5

```
word = "Python"  
print(word[0]) # 'P' – перший символ  
print(word[2]) # 't' – третій символ  
print(word[5]) # 'n' – останній символ
```

```
# Якщо спробувати індекс за межами – отримаємо помилку  
# print(word[10]) # IndexError: string index out of range
```

Але Python має чудову особливість – **негативні індекси**. Вони відраховують з кінця рядка, починаючи з -1.

Рядок: P y t h o n
Індекс: -6 -5 -4 -3 -2 -1

```
word = "Python"  
print(word[-1]) # 'n' – останній символ  
print(word[-2]) # 'o' – передостанній
```

```
print(word[-6]) # 'P' – перший символ (він же -6)
```

Це надзвичайно зручно, коли потрібно отримати, наприклад, останній символ, не знаючи довжини рядка.

Незмінність рядків (immutability)

Це, мабуть, найважливіша властивість рядків, яку потрібно засвоїти. **Рядки в Python є незмінними.** Це означає, що ви не можете змінити окремий символ у вже існуючому рядку.

```
word = "Python"
```

```
# word[0] = "J" # ПОМИЛКА! TypeError: 'str' object does not support item assignment
```

Чому так зроблено? Незмінність дає кілька переваг:

- **Безпека:** рядки можна безпечно використовувати як ключі в словниках.
- **Ефективність:** Python може оптимізувати роботу з пам'яттю.
- **Простота:** легше передбачити поведінку програми.

Але що робити, якщо нам потрібно змінити рядок? Ми не змінюємо старий рядок, а створюємо **новий** на основі старого:

```
word = "Python"
```

```
new_word = "J" + word[1:] # створюємо новий рядок "Jython"
```

```
print(new_word) # Jython
```

```
print(word) # Python – старий рядок не змінився!
```

Це важливо розуміти: будь-які операції з рядками (зміна регістру, заміна символів, конкатенація) створюють **нові рядки**, не змінюючи оригінал.

```
text = "Hello"
```

```
upper_text = text.upper()
```

```
print(text) # Hello (оригінал не змінився)
```

```
print(upper_text) # HELLO (новий рядок)
```

Практичний приклад: аналіз тексту

```
# Програма для аналізу введеного тексту
```

```
text = input("Введіть текст: ")
```

```
print("\n██ АНАЛІЗ ТЕКСТУ")
```

```
print("=" * 40)
```

```

print(f'Довжина тексту: {len(text)} символів')
print(f'Перший символ: '{text[0]}'')
print(f'Останній символ: '{text[-1]}'')

# Підрахунок голосних (українські)
vowels = "аеіоіуяюєїАЕИОІУЯЮЄЇ"
vowel_count = 0
for char in text:
    if char in vowels:
        vowel_count += 1

print(f'Кількість голосних: {vowel_count}')
print(f'Кількість приголосних: {len(text) - vowel_count}')

# Перевірка, чи є текст паліндромом (читається однаково з обох боків)
# Ми поки що не вміємо перевертати рядки, але скоро навчимося!

```

2. Зрізи (slicing): мистецтво виділення підрядків

Зрізи (slices) – це одна з найпотужніших та найелегантніших особливостей Python. Вони дозволяють отримувати підрядки (або підпоследовності) за допомогою дуже простого та зрозумілого синтаксису.

Базовий синтаксис зрізів

Зріз записується у квадратних дужках і має формат [start:stop:step]:

- start – індекс, з якого починається зріз (включно)
- stop – індекс, на якому зріз закінчується (не включно!)
- step – крок (з якою частотою брати символи)

Усі параметри є необов'язковими.

```
text = "Програмування"
```

Отримуємо підрядок з індексу 0 до 8 (не включаючи 8)

```
print(text[0:8]) # Програм
```

Якщо пропустити start – починаємо з початку

```
print(text[:8]) # Програм (the same)
```

```
# Якщо пропустити stop – йдемо до кінця  
print(text[8:]) # ування
```

Чому stop не включається?

Це може здаватися нелогічним, але насправді дуже зручно:

- Довжина зрізу = stop - start (якщо step = 1)
- Легко комбінувати зрізи: text[:5] і text[5:] дають неперетинні частини, які

разом складають весь рядок

```
text = "Python"  
first_part = text[:3] # Pyt  
second_part = text[3:] # hon  
print(first_part + second_part) # Python
```

Негативні індекси у зрізах

Негативні індекси теж працюють, і це дуже зручно:

```
text = "Програмування"
```

```
# Останні 5 символів  
print(text[-5:]) # ання
```

```
# Все, крім останніх 3 символів  
print(text[:-3]) # Програмува
```

```
# З 3-го символу з кінця до кінця  
print(text[-3:]) # ння
```

Крок (step) – третій вимір зрізів

Крок визначає, з яким інтервалом брати символи:

```
text = "Програмування"
```

```
# Кожен другий символ  
print(text[::2]) # Пгрмвн
```

```
# Кожен третій символ  
print(text[::3]) # Пгв
```

Зворотний порядок (крок -1) – перевертання рядка!

```
print(text[::-1]) # яиннавумаргорп
```

Це дуже елегантний спосіб перевернути рядок!

Комбінації start, stop, step

```
text = "Програмування"
```

З індексу 2 до 10 з кроком 2

```
print(text[2:10:2]) # оруа
```

З індексу -8 до -2 з кроком 1

```
print(text[-8:-2]) # уванн
```

З індексу 10 до 2 з кроком -1 (зворотній напрямок)

```
print(text[10:2:-1]) # аннвामур
```

Зрізи за межами рядка

На відміну від звичайної індексації, зрізи пробачають вихід за межі:

```
text = "Python"
```

```
print(text[0:100]) # Python – просто обрізає до кінця
```

```
print(text[-100:3]) # Pyt – починає з початку
```

```
print(text[100:200]) # "" – порожній рядок, не помилка!
```

Практичне застосування зрізів

1. Отримання домену з email:

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

```
at_index = email.find("@")
```

```
domain = email[at_index + 1:]
```

```
print(domain) # example.com
```

Або ще простіше, якщо знаємо структуру

```
username, domain = email.split("@")
```

```
print(username) # user
```

```
print(domain) # example.com
```

2. Перевірка на паліндром:

```
def is_palindrome(word):
    # Очищаємо слово (забираємо пробіли, переводимо в нижній регістр)
    word = word.replace(" ", "").lower()
    return word == word[::-1]
```

```
test_words = ["radar", "hello", "level", "Python", "мадам"]
for word in test_words:
    if is_palindrome(word):
        print(f'{word} – паліндром')
    else:
        print(f'{word} – не паліндром')
```

3. Форматування номера телефону:

```
phone = "380501234567"
formatted = f'+{phone[0:2]} ({phone[2:5]}) {phone[5:8]}-{phone[8:10]}-{phone[10:]}'
print(formatted) # +38 (050) 123-45-67
```

4. Отримання розширення файлу:

```
filename = "document.pdf"
dot_index = filename.rfind(".")
if dot_index != -1:
    extension = filename[dot_index + 1:]
    name = filename[:dot_index]
    print(f'Ім'я: {name}, розширення: {extension}')
```

Практичний приклад: текстовий процесор

Об'єднаємо знання про індексацію та зрізи для створення програми, яка імітує прості операції текстового редактора:

```
print("🖨️ ПРОСТИЙ ТЕКСТОВИЙ ПРОЦЕСОР")
print("=" * 50)
```

Вихідний текст

```
text = "Python – це потужна мова програмування"
print(f'Текст: {text}\n')
```

```

while True:
    print("\nДоступні операції:")
    print("1. Показати символ за індексом")
    print("2. Показати підрядок (зріз)")
    print("3. Перевернути текст")
    print("4. Показати довжину тексту")
    print("5. Вийти")

    choice = input("Оберіть операцію (1-5): ")

    if choice == "1":
        try:
            index = int(input("Введіть індекс: "))
            print(f"Символ на позиції {index}: '{text[index]}'")
        except IndexError:
            print("✗ Помилка: індекс за межами рядка")
        except ValueError:
            print("✗ Помилка: введіть число")

    elif choice == "2":
        print("Введіть параметри зрізу (enter – пропустити):")
        start = input("start: ")
        stop = input("stop: ")
        step = input("step: ")

        # Перетворюємо порожні рядки на None
        start = int(start) if start else None
        stop = int(stop) if stop else None
        step = int(step) if step else None

        try:
            slice_text = text[start:stop:step]
            print(f"Результат: '{slice_text}'")
        except Exception as e:
            print(f"✗ Помилка: {e}")

    elif choice == "3":

```

```

print(f"Оригінал: {text}")
print(f"Перевернуто: {text[::-1]}")

elif choice == "4":
    print(f"Довжина тексту: {len(text)} символів")

elif choice == "5":
    print("До побачення!")
    break

else:
    print("✗ Невірний вибір")

print("-" * 50)

```

Типові помилки при роботі з рядками

1. Забування про незмінність:

```

text = "hello"
text.upper() # здається, що змінили, але...
print(text) # все ще "hello"!

```

Правильно:

```
text = text.upper()
```

2. Плутанина з індексами:

```

text = "Python"
# Останній символ має індекс 5, або -1
# text[6] – помилка!

```

3. Спроба змінити символ:

```

text = "Python"
# text[0] = "J" # TypeError!
# Правильно:
text = "J" + text[1:]

```

У наступних пунктах ми розглянемо багаті можливості методів рядків, які дозволяють виконувати пошук, заміну, перевірку та форматування тексту.

3. Основні методи рядків: трансформація, пошук та перевірка

Рядки в Python мають безліч корисних методів. Важливо пам'ятати: оскільки рядки незмінні, всі методи, які "змінюють" рядок, насправді повертають новий рядок, залишаючи оригінал без змін.

Методи зміни регістру

Найчастіше використовувані методи – для роботи з регістром літер:

```
text = "Python Programming"
```

```
print(text.upper())      # PYTHON PROGRAMMING – всі великі
```

```
print(text.lower())      # python programming – всі малі
```

```
print(text.capitalize()) # Python programming – перша велика, решта малі
```

```
print(text.title())      # Python Programming – кожне слово з великої
```

```
print(text.swapcase())   # pYTHON pROGRAMMING – зміна регістру на  
протилежний
```

```
print(text) # Python Programming – оригінал не змінився!
```

Ці методи дуже корисні для нормалізації введених даних. Наприклад, коли користувач може ввести ім'я в різному регістрі:

```
name = input("Введіть ваше ім'я: ").strip().title()
```

```
print(f"Привіт, {name}!")
```

Методи очищення рядка

Коли ми отримуємо дані від користувача або з файлу, часто бувають зайві пробіли на початку або в кінці:

```
text = "  багато пробілів навколо  "
```

```
print(f"{'{text}'}")
```

```
print(f"{'{text.strip()}'}") # 'багато пробілів навколо' – з обох боків
```

```
print(f"{'{text.lstrip()}'}") # 'багато пробілів навколо' – тільки зліва
```

```
print(f"{'{text.rstrip()}'}") # 'багато пробілів навколо' – тільки справа
```

Можна видаляти не тільки пробіли, а й інші символи

```
website = "www.python.org"
```

```
print(website.strip("w"))    # .python.org
print(website.strip("w.org")) # www.python (увага: видаляє будь-які з цих
символів з обох боків)
```

Методи пошуку в рядку

Python пропонує кілька методів для пошуку підрядків:

```
text = "Python – це потужна мова програмування. Python дуже
популярний."
```

find() – повертає індекс першого входження або -1, якщо не знайдено

```
print(text.find("потужна"))    # 12 (індекс початку слова)
print(text.find("Java"))       # -1 (не знайдено)
```

rfind() – пошук справа (повертає останнє входження)

```
print(text.rfind("Python"))    # 36 (друге слово Python)
```

index() – як find(), але генерує помилку, якщо не знайдено

```
print(text.index("потужна"))    # 12
# print(text.index("Java"))     # ValueError! Будьте обережні
```

count() – кількість входжень

```
print(text.count("Python"))    # 2
print(text.count("н"))         # 5 (літера "н" зустрічається 5 разів)
```

startswith() / endswith() – перевірка початку/кінця

```
print(text.startswith("Python")) # True
print(text.endswith("."))        # True
```

Методи перевірки вмісту рядка

Ці методи повертають True або False і часто використовуються для валідації даних:

Перевірка, з чого складається рядок

```
print("12345".isdigit())    # True – тільки цифри
print("123abc".isdigit())   # False – є букви
print("abc".isalpha())      # True – тільки букви
print("abc123".isalnum())   # True – букви та/або цифри
```

```

print("abc123".isalnum())    # True (немає спецсимволів)
print(" ".isspace())        # True – тільки пробільні символи

# Перевірка регістру
print("HELLO".isupper())    # True
print("hello".islower())    # True
print("Hello World".istitle()) # True – кожне слово з великої

# Інші перевірки
print("Hello\n".isprintable()) # False – є символ нового рядка

```

Практичний приклад: валідація даних користувача

```

def validate_user_data():
    print("👤 РЕЄСТРАЦІЯ КОРИСТУВАЧА")
    print("=" * 40)

    # Валідація імені
    while True:
        name = input("Ім'я: ").strip()
        if not name:
            print("❌ Ім'я не може бути порожнім")
        elif not name.replace(" ", "").isalpha():
            print("❌ Ім'я має містити тільки літери")
        elif len(name) < 2:
            print("❌ Ім'я занадто коротке")
        else:
            name = name.title()
            break

    # Валідація email
    while True:
        email = input("Email: ").strip().lower()
        if not email:
            print("❌ Email не може бути порожнім")
        elif email.count("@") != 1:
            print("❌ Email має містити один символ @")

```

```

elif not email.endswith((".com", ".ua", ".org", ".net")):
    print("✗ Email має закінчуватися на .com, .ua, .org або .net")
else:
    break

# Валідація віку
while True:
    age = input("Вік: ").strip()
    if not age.isdigit():
        print("✗ Вік має бути числом")
    else:
        age = int(age)
        if age < 18 or age > 120:
            print("✗ Вік має бути від 18 до 120")
        else:
            break

# Валідація пароля
while True:
    password = input("Пароль: ")
    if len(password) < 8:
        print("✗ Пароль має містити хоча б 8 символів")
    elif not any(c.isupper() for c in password):
        print("✗ Пароль має містити хоча б одну велику літеру")
    elif not any(c.islower() for c in password):
        print("✗ Пароль має містити хоча б одну малу літеру")
    elif not any(c.isdigit() for c in password):
        print("✗ Пароль має містити хоча б одну цифру")
    else:
        break

print("\n✓ РЕЄСТРАЦІЯ УСПІШНА!")
print(f"Ім'я: {name}")
print(f"Email: {email}")
print(f"Вік: {age}")
print(f"=" * 40)

```

```
# validate_user_data()
```

Методи заміни та розширені можливості

```
# replace() – заміна підрядка
```

```
text = "Я люблю Python. Python це круто!"
```

```
new_text = text.replace("Python", "Java")
```

```
print(new_text) # Я люблю Java. Java це круто!
```

```
# Можна обмежити кількість заміни
```

```
new_text = text.replace("Python", "Java", 1) # тільки перше входження
```

```
print(new_text) # Я люблю Java. Python це круто!
```

```
# expandtabs() – заміна табуляцій пробілами
```

```
text_with_tabs = "a\tb\tc"
```

```
print(text_with_tabs.expandtabs(10)) # a      b      c
```

4. Розбиття, з'єднання та форматування рядків

Одні з найпотужніших операцій з рядками – це перетворення рядків на списки (розбиття) і навпаки (з'єднання). Це основа для обробки текстових даних.

Розбиття рядка: `split()` та `splitlines()`

Метод `split()` розбиває рядок на список підрядків за вказаним роздільником:

```
# Розбиття за пробілами (за замовчуванням)
```

```
sentence = "Python це потужна мова програмування"
```

```
words = sentence.split()
```

```
print(words) # ['Python', 'це', 'потужна', 'мова', 'програмування']
```

```
# Розбиття за конкретним роздільником
```

```
data = "Іваненко,25,Київ,студент"
```

```
fields = data.split(",")
```

```
print(fields) # ['Іваненко', '25', 'Київ', 'студент']
```

```
# Обмеження кількості розбиттів
text = "a:b:c:d:e"
print(text.split(":", 2)) # ['a', 'b', 'c:d:e'] – тільки перші два роздільники
```

```
# splitlines() – розбиття за рядками
multiline = "Перший рядок\nДругий рядок\nТретій рядок"
lines = multiline.splitlines()
print(lines) # ['Перший рядок', 'Другий рядок', 'Третій рядок']
```

З'єднання рядків: join()

Метод join() – це, мабуть, найефективніший спосіб об'єднати багато рядків в один. Він викликається від **роздільника** і приймає список рядків:

```
# Об'єднання списку слів у речення
words = ['Python', 'це', 'потужна', 'мова']
sentence = ' '.join(words)
print(sentence) # Python це потужна мова
```

```
# Інші роздільники
tags = ['python', 'programming', 'coding']
hashtags = '#'.join(tags)
print(hashtags) # python#programming#coding
```

```
# Об'єднання з переходом на новий рядок
lines = ['Рядок 1', 'Рядок 2', 'Рядок 3']
text = '\n'.join(lines)
print(text)
```

Чому join() краще за конкатенацію?

При роботі з великою кількістю рядків використання + у циклі створює багато проміжних рядків, що дуже неефективно. join() працює набагато швидше:

```
# ПОГАНО (для великих даних)
result = ""
for word in words:
    result += word + " " # створюється новий рядок на кожній ітерації
```

ДОБРЕ

result = " ".join(words) *# один прохід, один новий рядок*

Практичний приклад: парсинг CSV-файлу

Імітація даних з CSV-файлу

csv_data = """"Іваненко,25,Київ,студент

Петренко,30,Львів,програміст

Сидоренко,22,Харків,дизайнер

Коваленко,35,Одеса,менеджер"""

print("██ ПАРСИНГ CSV-ДАНИХ")

print("=" * 50)

Розбиваємо на рядки

lines = csv_data.splitlines()

print(f"Знайдено {len(lines)} записів\n")

Обробляємо кожен рядок

for i, line in enumerate(lines, 1):

fields = line.split(",")

name, age, city, occupation = fields

print(f"Запис #{i}:")

print(f" Ім'я: {name}")

print(f" Вік: {age}")

print(f" Місто: {city}")

print(f" Професія: {occupation}")

print()

Створюємо новий CSV з іншим роздільником

new_csv = []

for line in lines:

fields = line.split(",")

Змінюємо порядок полів

new_line = "|".join([fields[0], fields[3], fields[2], fields[1]])

new_csv.append(new_line)

```
print("НОВИЙ ФОРМАТ (з роздільником |):")
print("\n".join(new_csv))
print("=" * 50)
```

Форматування рядків: від минулого до сучасності

Python пропонує кілька способів форматування рядків. Ми зосередимося на сучасному та найкращому способі – **f-strings**, але згадаємо й інші для розуміння старого коду.

1. Оператор % (старий стиль, як у C):

```
name = "Олена"
age = 20
print("Мене звати %s, мені %d років" % (name, age))
```

%s – рядок, %d – ціле число, %f – дробове

2. Метод format() (стиль Python 3.0+):

```
print("Мене звати {}, мені {} років".format(name, age))
print("Мене звати {0}, мені {1} років, {0} – гарне ім'я".format(name, age))
print("Мене звати {name}, мені {age} років".format(name=name, age=age))
```

3. f-strings (Python 3.6+, сучасний стандарт):

```
print(f"Мене звати {name}, мені {age} років")
```

Можливості f-strings

f-strings надзвичайно потужні та гнучкі:

```
name = "Марія"
age = 19
city = "Львів"
pi = 3.14159265359
```

Обчислення всередині {}

```
print(f"Через 5 років {name} буде {age + 5} років")
```

Виклик методів

```
print(f"Ім'я у верхньому регістрі: {name.upper()}")
```


Форматування чисел

```
print(f"Число π з двома знаками: {pi:.2f}")  
print(f"Число π з чотирма знаками: {pi:.4f}")
```

Вирівнювання

```
print(f"|{'текст':<10}|") # вирівнювання вліво  
print(f"|{'текст':>10}|") # вирівнювання вправо  
print(f"|{'текст':^10}|") # центрування
```

Великі вирази

```
x, y = 10, 3  
print(f"{x} + {y} = {x + y}")  
print(f"{x} / {y} = {x / y:.2f}")
```

Багаторядкові f-strings

```
user = {  
    "name": "Іван",  
    "email": "ivan@example.com",  
    "age": 28,  
    "city": "Київ",  
    "occupation": "програміст"  
}
```

```
profile = f"""
```

```
    ПРОФІЛЬ    ||  
    ||  
    ||  
    || Ім'я:   {user['name']:<12} ||  
    || Email: {user['email']:<12} ||  
    || Вік:   {user['age']:<12} ||  
    || Місто: {user['city']:<12} ||  
    || Робота: {user['occupation']:<12} ||  
    ||  
    ||
```

```
    """
```

```
print(profile)
```

Практичний приклад: генератор листів

```

def generate_letter(template, **kwargs):
    """Генерує лист, підставляючи значення в шаблон"""
    return template.format(**kwargs)

# Шаблон листа (використовуємо старий format для гнучкості)
letter_template = """
Шановний(а) {name}!

Дякуємо за реєстрацію на нашому сайті {site}.
Ваш логін: {login}
Ваш тимчасовий пароль: {password}

Для підтвердження реєстрації перейдіть за посиланням:
{site}/confirm?code={code}

З повагою,
Команда {site}
"""

# Дані для різних користувачів
users = [
    {"name": "Олена Петренко", "login": "olena_p", "password": "qwerty123",
"code": "ABC123"},
    {"name": "Іван Коваленко", "login": "ivan_k", "password": "pass456",
"code": "XYZ789"},
    {"name": "Марія Шевченко", "login": "maria_s", "password": "maria2024",
"code": "DEF456"},
]

# Генеруємо листи
for user_data in users:
    letter = generate_letter(letter_template, site="python-courses.ua",
**user_data)
    print(letter)
    print("=" * 60)

```

Підсумковий практичний приклад: аналізатор тексту

Об'єднаємо всі знання про рядки в одну програму:

```
print("🔍 АНАЛІЗАТОР ТЕКСТУ")
print("=" * 60)

# Отримуємо текст від користувача
text = input("Введіть текст: ").strip()

if not text:
    print("❌ Текст не може бути порожнім")
else:
    # Основна статистика
    print(f"\n📊 СТАТИСТИКА:")
    print(f"Загальна довжина: {len(text)} символів")
    print(f"Кількість символів без пробілів: {len(text.replace(' ', ''))}")

    # Кількість слів
    words = text.split()
    print(f"Кількість слів: {len(words)}")

    # Аналіз регістру
    upper_count = sum(1 for c in text if c.isupper())
    lower_count = sum(1 for c in text if c.islower())
    digit_count = sum(1 for c in text if c.isdigit())
    space_count = sum(1 for c in text if c.isspace())

    print(f"Великих літер: {upper_count}")
    print(f"Малих літер: {lower_count}")
    print(f"Цифр: {digit_count}")
    print(f"Пробілів: {space_count}")

    # Знаходимо найдовше слово
    longest_word = max(words, key=len) if words else ""
    print(f"Найдовше слово: '{longest_word}' ({len(longest_word)} символів)")

    # Перевіряємо, чи текст є числом
```

```
if text.replace(".", "").replace("-", "").replace(",", "").isdigit():
    print("Текст складається тільки з цифр")
```

Заміна слів

```
if len(words) > 2:
    old_word = input("\nЯке слово замінити? ")
    new_word = input("На яке слово замінити? ")

    if old_word in text:
        new_text = text.replace(old_word, new_word)
        print(f"\nРезультат заміни:")
        print(new_text)
    else:
        print(f"Слово '{old_word}' не знайдено")
```

Ефект друкарської машинки (виведення по словах)

```
print("\n🖨 Текст по словах:")
for i, word in enumerate(words, 1):
    print(f"{i:2}. {word}")
```

Створюємо хештеги

```
hashtags = ['#' + word.strip('.,!?:;').lower() for word in words if len(word) >
```

3]

```
if hashtags:
    print(f"\n🏷 Хештеги: {' '.join(hashtags[:5])}")
```

Форматуємо текст для виведення

```
print("\n📄 Відформатований текст:")
print(f"Перші 50 символів: {text[:50]}...")
print(f"Останні 50 символів: {text[-50:]}")
print(f"Кожне друге слово: {' '.join(words[::2])}")
```

```
print("=" * 60)
```

Ця програма демонструє практично всі вивчені методи роботи з рядками: розбиття, пошук, заміну, перевірку, форматування та багато іншого.

У наступній лекції ми перейдемо до списків та кортежів – перших "справжніх" структур даних, які дозволяють зберігати колекції елементів.

Питання для обговорення

1. Чому рядки в Python називають незмінними (immutable)? Що станеться, якщо спробувати змінити окремий символ рядка? Як тоді можна отримати рядок із зміненим символом?
2. Поясніть різницю між позитивною та негативною індексацією в рядках. Чому негативні індекси корисні? Як отримати останній символ рядка, не знаючи його довжини?
3. Що таке зріз (slicing) рядка? Поясніть синтаксис [start:stop:step]. Що буде результатом виконання "Програмування"[::-1]? Як це працює?
4. Проаналізуйте код: `text = "Python"; text.upper(); print(text)`. Чому на екрані з'явиться "Python", а не "PYTHON"? Як правильно отримати рядок у верхньому регістрі?
5. Поясніть різницю між методами `find()` та `index()`. У яких ситуаціях краще використовувати `find()`, а в яких `index()`? Чому?
6. Що роблять методи `split()` та `join()`? Наведіть приклад, де ці методи використовуються разом для обробки рядка. Чому `join()` ефективніший за конкатенацію у циклі?
7. Як перевірити, чи складається рядок тільки з цифр? Як перевірити, чи рядок починається з певного слова? Які методи для цього використовуються?
8. Напишіть програму, яка запитує у користувача email і перевіряє, чи містить він символ @ та чи закінчується на .com, .ua або .org. Використовуйте методи рядків.
9. Що таке f-рядки? Чим вони кращі за старий метод форматування `.format()`? Наведіть приклад використання f-рядка з обчисленням виразу всередині фігурних дужок.
10. Проаналізуйте код: `print("Hello" + 5)`. Чому виникає помилка? Як її виправити двома різними способами (з використанням `str()` та використанням f-рядка)?

ЛЕКЦІЯ 7

Списки та кортежі

План лекції

1. Списки як упорядковані колекції: створення, індексація та змінність.

Почнемо з основ: що таке список, як його створити, як звертатися до елементів. Головний акцент зробимо на ключовій відмінності списків від рядків – змінності (mutability). Розберемо індексацію та зрізи для списків, показавши схожість з рядками.

2. Методи списків: додавання, видалення та модифікація елементів.

Детально розглянемо всі основні методи для роботи зі списками: `append()`, `extend()`, `insert()` для додавання; `remove()`, `pop()`, `clear()` для видалення; а також корисні методи `sort()`, `reverse()`, `index()`, `count()`. Покажемо, як ефективно маніпулювати списками.

3. Кортежі – незмінні брати списків.

Познайомимось з кортежами, їх синтаксисом та особливостями. Пояснимо, чому незмінність – це не недолік, а перевага в певних ситуаціях. Розберемо, коли варто використовувати кортежі замість списків.

4. Вкладені списки та розпакування.

Покажемо, як створювати списки списків (матриці) та працювати з ними. Розберемо потужний механізм розпакування (unpacking), який дозволяє елегантно присвоювати значення зі списків та кортежів окремим змінним.

1. Списки як упорядковані колекції: створення, індексація та змінність

Досі ми працювали з простими типами даних: числами, рядками, булевими значеннями. Вони зберігають одне значення. Але в реальному житті ми маємо справу з колекціями: список студентів у групі, перелік температур за тиждень, кошик покупок в інтернет-магазині. Саме для таких випадків у Python існують **списки (lists)**.

Що таке список?

Список – це впорядкована, змінювана колекція елементів, яка може містити дані різних типів. У Python список створюється за допомогою квадратних дужок `[]`, а елементи розділяються комами.

Порожній список

```
empty_list = []
```

```
# Список чисел
```

```
numbers = [1, 2, 3, 4, 5]
```

```
# Список рядків
```

```
fruits = ["яблуко", "банан", "апельсин", "груша"]
```

```
# Змішаний список (різні типи даних)
```

```
mixed = [42, "hello", 3.14, True, None]
```

```
# Список списків (вкладений список)
```

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
print(f"Порожній: {empty_list}")
```

```
print(f"Числа: {numbers}")
```

```
print(f"Фрукти: {fruits}")
```

```
print(f"Змішаний: {mixed}")
```

```
print(f"Матриця: {matrix}")
```

Створення списку за допомогою функції list()

Крім літералів, списки можна створювати за допомогою вбудованої функції list():

```
# З рядка – отримуємо список символів
```

```
chars = list("Python")
```

```
print(chars) # ['P', 'y', 't', 'h', 'o', 'n']
```

```
# З діапазону
```

```
numbers = list(range(1, 6))
```

```
print(numbers) # [1, 2, 3, 4, 5]
```

```
# Порожній список
```

```
empty = list()
```

```
print(empty) # []
```

Довжина списку

Як і для рядків, для списків можна отримати довжину за допомогою функції `len()`:

```
fruits = ["яблуко", "банан", "апельсин", "груша"]  
print(f"Кількість фруктів: {len(fruits)}") # 4
```

Індексація списків

До елементів списку можна звертатися за індексом, точно так само, як до символів рядка. Індеси починаються з 0.

```
fruits = ["яблуко", "банан", "апельсин", "груша"]
```

```
print(fruits[0]) # яблуко  
print(fruits[1]) # банан  
print(fruits[3]) # груша
```

Негативні індекси – з кінця

```
print(fruits[-1]) # груша  
print(fruits[-2]) # апельсин
```

Якщо індекс за межами – помилка

```
# print(fruits[10]) # IndexError: list index out of range
```

Зрізи (slicing) для списків

Зрізи працюють зі списками так само, як з рядками:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
print(numbers[2:7]) # [2, 3, 4, 5, 6]  
print(numbers[:5]) # [0, 1, 2, 3, 4]  
print(numbers[5:]) # [5, 6, 7, 8, 9]  
print(numbers[::2]) # [0, 2, 4, 6, 8]  
print(numbers[::-1]) # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] – перевернутий список  
print(numbers[-5:]) # [5, 6, 7, 8, 9] – останні 5 елементів
```

Головна відмінність від рядків: змінність (mutability)

Це найважливіша властивість списків. На відміну від рядків, які є незмінними, списки можна **змінювати**. Ми можемо змінити окремий елемент, додати новий або видалити існуючий.

Рядки – незмінні

```
name = "Python"
```

name[0] = "J" # ПОМИЛКА!

Списки – змінні

```
fruits = ["яблуко", "банан", "апельсин"]
```

```
print(f"Початковий: {fruits}")
```

Змінюємо елемент за індексом

```
fruits[1] = "груша"
```

```
print(f"Після зміни: {fruits}") # ['яблуко', 'груша', 'апельсин']
```

Змінюємо зріз

```
numbers = [1, 2, 3, 4, 5]
```

```
numbers[1:4] = [20, 30, 40]
```

```
print(numbers) # [1, 20, 30, 40, 5]
```

Можна навіть вставити більше або менше елементів

```
numbers = [1, 2, 3, 4, 5]
```

```
numbers[1:4] = [20, 30] # замінюємо 3 елементи двома
```

```
print(numbers) # [1, 20, 30, 5]
```

Ітерування по списку

Як і будь-яку послідовність, список можна обходити в циклі:

```
fruits = ["яблуко", "банан", "апельсин", "груша"]
```

Простий цикл

```
for fruit in fruits:
```

```
    print(f"Смачний фрукт: {fruit}")
```

Якщо потрібні індекси

```
for i, fruit in enumerate(fruits):
```

```
    print(f"{i}: {fruit}")
```

Перевірка наявності елемента

Оператор in дозволяє перевірити, чи є елемент у списку:

```
fruits = ["яблуко", "банан", "апельсин", "груша"]
```

```
if "банан" in fruits:  
    print("Банан є в списку!")  
else:  
    print("Банан не знайдено")
```

```
if "ананас" not in fruits:  
    print("Ананаса немає, але можна додати!")
```

Практичний приклад: управління списком справ

```
# Простий менеджер списку справ
```

```
tasks = []
```

```
print("📁 МЕНЕДЖЕР СПРАВ")  
print("=" * 40)
```

```
while True:  
    print(f"\nПоточні справи ({len(tasks)}):")  
    if tasks:  
        for i, task in enumerate(tasks, 1):  
            print(f"{i}. {task}")  
    else:  
        print("Список порожній")
```

```
print("\nКоманди:")  
print(" add - додати справу")  
print(" remove - видалити справу")  
print(" exit - вийти")
```

```
command = input("\nВведіть команду: ").strip().lower()
```

```
if command == "exit":  
    print("До побачення!")  
    break  
elif command == "add":
```

```

task = input("Введіть нову справу: ").strip()
if task:
    tasks.append(task)
    print(f"✓ Додано: {task}")
else:
    print("✗ Справа не може бути порожньою")
elif command == "remove":
    if not tasks:
        print("✗ Список порожній")
        continue

    try:
        index = int(input("Введіть номер справи для видалення: ")) - 1
        if 0 <= index < len(tasks):
            removed = tasks.pop(index)
            print(f"✓ Видалено: {removed}")
        else:
            print("✗ Невірний номер")
    except ValueError:
        print("✗ Введіть число")
else:
    print("✗ Невідома команда")

```

2. Методи списків: додавання, видалення та модифікація елементів

Python надає багатий набір методів для роботи зі списками. Розглянемо їх детально.

Методи додавання елементів

Початковий список

```

fruits = ["яблуко", "банан"]
print(f"Початковий: {fruits}")

```

append(x) – додає один елемент у кінець списку

```

fruits.append("апельсин")

```

```
print(f"Після append: {fruits}") # ['яблуко', 'банан', 'апельсин']
```

append зі списком – додасть список як один елемент

```
fruits.append(["груша", "слива"])
```

```
print(f"Після append списку: {fruits}") # ['яблуко', 'банан', 'апельсин',  
['груша', 'слива']]
```

extend(iterable) – додає всі елементи з іншого списку

```
more_fruits = ["груша", "слива"]
```

```
fruits = ["яблуко", "банан"] # починаємо заново
```

```
fruits.extend(more_fruits)
```

```
print(f"Після extend: {fruits}") # ['яблуко', 'банан', 'груша', 'слива']
```

insert(i, x) – вставляє елемент на позицію i

```
fruits.insert(1, "апельсин") # вставляємо на другу позицію (індекс 1)
```

```
print(f"Після insert: {fruits}") # ['яблуко', 'апельсин', 'банан', 'груша',  
'слива']
```

Важлива відмінність між append() та extend()

append – додає один об'єкт

```
list1 = [1, 2, 3]
```

```
list1.append([4, 5])
```

```
print(list1) # [1, 2, 3, [4, 5]] – список став довшим на 1 елемент
```

extend – додає елементи з послідовності

```
list2 = [1, 2, 3]
```

```
list2.extend([4, 5])
```

```
print(list2) # [1, 2, 3, 4, 5] – список став довшим на 2 елементи
```

Методи видалення елементів

```
fruits = ["яблуко", "банан", "апельсин", "груша", "слива"]
```

```
print(f"Початковий: {fruits}")
```

remove(x) – видаляє ПЕРШЕ входження елемента зі значенням x

```
fruits.remove("банан")
```

```
print(f"Після remove('банан'): {fruits}")
```

Якщо елемента немає – помилка

fruits.remove("ананас") # ValueError: list.remove(x): x not in list

pop(i) – видаляє та повертає елемент за індексом i (за замовчуванням останній)

`last = fruits.pop()` *# видаляє останній*

`print(f"Видалений останній: {last}")`

`print(f"Після pop(): {fruits}")`

`second = fruits.pop(1)` *# видаляє елемент з індексом 1*

`print(f"Видалений елемент з індексом 1: {second}")`

`print(f"Після pop(1): {fruits}")`

clear() – видаляє всі елементи

`fruits.clear()`

`print(f"Після clear(): {fruits}")` *# []*

Пошук та підрахунок

`numbers = [10, 20, 30, 20, 40, 20, 50]`

index(x) – повертає індекс ПЕРШОГО входження x

`print(numbers.index(20))` *# 1*

`print(numbers.index(40))` *# 4*

print(numbers.index(100)) # ValueError!

index(x, start, end) – пошук з певної позиції

`print(numbers.index(20, 2))` *# 3 (шукаємо 20, починаючи з індексу 2)*

count(x) – кількість входжень x

`print(numbers.count(20))` *# 3*

`print(numbers.count(100))` *# 0*

Сортування та реверсування

`numbers = [5, 2, 8, 1, 9, 3]`

sort() – сортує список на місці (змінює оригінал!)

```
numbers.sort()
```

```
print(f'Після sort(): {numbers}') # [1, 2, 3, 5, 8, 9]
```

sort(reverse=True) – сортування за спаданням

```
numbers.sort(reverse=True)
```

```
print(f'За спаданням: {numbers}') # [9, 8, 5, 3, 2, 1]
```

reverse() – перевертає список на місці

```
numbers.reverse()
```

```
print(f'Після reverse(): {numbers}') # знову [1, 2, 3, 5, 8, 9]
```

sorted() – вбудована функція, яка створює НОВИЙ відсортований список

```
original = [5, 2, 8, 1, 9, 3]
```

```
sorted_list = sorted(original)
```

```
print(f'Оригінал: {original}') # [5, 2, 8, 1, 9, 3] – не змінився!
```

```
print(f'Відсортований: {sorted_list}') # [1, 2, 3, 5, 8, 9]
```

sorted() з reverse=True

```
descending = sorted(original, reverse=True)
```

```
print(f'За спаданням: {descending}') # [9, 8, 5, 3, 2, 1]
```

Копіювання списків

Це дуже важливий момент! Просте присвоєння не створює копію, а лише нове посилання на той самий список:

```
list1 = [1, 2, 3]
```

```
list2 = list1 # це не копія! це те саме посилання
```

```
list2.append(4)
```

```
print(list1) # [1, 2, 3, 4] – змінилося! (бо це той самий список)
```

```
print(list2) # [1, 2, 3, 4]
```

Як створити справжню копію?

Спосіб 1: copy() метод

```
list1 = [1, 2, 3]
```

```
list2 = list1.copy()
```

```
list2.append(4)
print(list1) # [1, 2, 3] – не змінився!
print(list2) # [1, 2, 3, 4]
```

```
# Спосіб 2: list() конструктор
list1 = [1, 2, 3]
list2 = list(list1)
```

```
# Спосіб 3: зріз [:]
list1 = [1, 2, 3]
list2 = list1[:]
```

Але будьте обережні з вкладеними списками!

```
# Поверхнева копія (shallow copy)
original = [[1, 2], [3, 4]]
copy = original.copy()
```

```
copy[0][0] = 99
print(original) # [[99, 2], [3, 4]] – теж змінилося!
```

```
# Для глибокого копіювання потрібен модуль copy
import copy
original = [[1, 2], [3, 4]]
deep_copy = copy.deepcopy(original)
```

```
deep_copy[0][0] = 99
print(original) # [[1, 2], [3, 4]] – не змінилося!
print(deep_copy) # [[99, 2], [3, 4]]
```

Інші корисні методи та функції

```
numbers = [3, 1, 4, 1, 5, 9, 2]
```

```
# len() – довжина списку
print(f'Довжина: {len(numbers)}')
```

```
# min(), max() – мінімум та максимум
```

```

print(f'Мінімум: {min(numbers)}')
print(f'Максимум: {max(numbers)}')

# sum() – сума (працює для чисел)
print(f'Сума: {sum(numbers)}')

# any() – чи є хоч один істинний елемент
print(any([False, False, True])) # True
print(any([0, "", None]))        # False

# all() – чи всі елементи істинні
print(all([True, True, True]))   # True
print(all([1, 2, 0, 4]))         # False (0 – це False)

```

Практичний приклад: статистика оцінок

```

# Програма для обробки оцінок студентів
grades = []

print("\n ОБРОБНИК ОЦІНОК")
print("=" * 50)
print("Вводьте оцінки (0-100). Для завершення введіть -1")

while True:
    try:
        grade = int(input("Оцінка: "))
        if grade == -1:
            break
        elif 0 <= grade <= 100:
            grades.append(grade)
        else:
            print("✗ Оцінка має бути від 0 до 100")
    except ValueError:
        print("✗ Введіть число")

if not grades:
    print("✗ Не введено жодної оцінки")

```


else:

```
print("\n📊 СТАТИСТИКА")
print(f"Кількість оцінок: {len(grades)}")
print(f"Мінімум: {min(grades)}")
print(f"Максимум: {max(grades)}")
print(f"Середнє: {sum(grades) / len(grades):.2f}")
```

Сортиємо

```
grades.sort()
print(f"\nВідсортовані оцінки: {grades}")
```

Знаходимо медіану

```
n = len(grades)
if n % 2 == 1:
    median = grades[n // 2]
else:
    median = (grades[n // 2 - 1] + grades[n // 2]) / 2
print(f"Медіана: {median}")
```

Найпоширеніша оцінка (мода)

```
from collections import Counter
counter = Counter(grades)
most_common = counter.most_common(1)
print(f"Найчастіша оцінка: {most_common[0][0]} (зустрічається {most_common[0][1]} разів)")
```

Розподіл за категоріями

```
excellent = sum(1 for g in grades if g >= 90)
good = sum(1 for g in grades if 75 <= g < 90)
satisfactory = sum(1 for g in grades if 60 <= g < 75)
poor = sum(1 for g in grades if g < 60)
```

```
print("\n📊 РОЗПОДІЛ:")
print(f"Відмінно (90+): {excellent}")
print(f"Добре (75-89): {good}")
print(f"Задовільно (60-74): {satisfactory}")
print(f"Незадовільно (<60): {poor}")
```

```
print("=" * 50)
```

Практичний приклад: список контактів

Проста адресна книга

```
contacts = []
```

```
print(" 📁 АДРЕСНА КНИГА")
```

```
print("=" * 50)
```

```
while True:
```

```
    print("\nКоманди:")
```

```
    print(" 1. Додати контакт")
```

```
    print(" 2. Показати всі контакти")
```

```
    print(" 3. Пошук контакту")
```

```
    print(" 4. Видалити контакт")
```

```
    print(" 5. Вийти")
```

```
choice = input("Оберіть команду (1-5): ").strip()
```

```
if choice == "1":
```

```
    name = input("Ім'я: ").strip().title()
```

```
    phone = input("Телефон: ").strip()
```

```
    email = input("Email: ").strip().lower()
```

```
    contact = [name, phone, email]
```

```
    contacts.append(contact)
```

```
    print(f"✅ Контакт {name} додано!")
```

```
elif choice == "2":
```

```
    if not contacts:
```

```
        print("📁 Адресна книга порожня")
```

```
    else:
```

```
        print("\n📁 СПИСОК КОНТАКТІВ:")
```

```
        for i, contact in enumerate(contacts, 1):
```

```
            print(f"{i}. {contact[0]} – {contact[1]} – {contact[2]}")
```

```

elif choice == "3":
    if not contacts:
        print("📖 Адресна книга порожня")
        continue

    search = input("Введіть ім'я для пошуку: ").strip().title()
    found = False
    for contact in contacts:
        if search in contact[0]: # частковий збіг
            print(f"✅ Знайдено: {contact[0]} – {contact[1]} – {contact[2]}")
            found = True
    if not found:
        print(f"❌ Контактів з іменем '{search}' не знайдено")

elif choice == "4":
    if not contacts:
        print("📖 Адресна книга порожня")
        continue

    print("\n📁 СПИСОК КОНТАКТІВ:")
    for i, contact in enumerate(contacts, 1):
        print(f"{i}. {contact[0]}")

    try:
        idx = int(input("Введіть номер контакту для видалення: ")) - 1
        if 0 <= idx < len(contacts):
            removed = contacts.pop(idx)
            print(f"✅ Контакт {removed[0]} видалено")
        else:
            print("❌ Невірний номер")
    except ValueError:
        print("❌ Введіть число")

elif choice == "5":
    print("До побачення!")

```

```
break
```

```
else:
```

```
print("✗ Невідома команда")
```

```
print("-" * 30)
```

У наступному пункті ми розглянемо кортежі – незмінні "брати" списків, які мають свої особливі застосування.

3. Кортежі – незмінні брати списків

Кортежі (tuples) – це ще один тип впорядкованих колекцій у Python. Вони дуже схожі на списки, але мають одну ключову відмінність: **кортежі є незмінними (immutable)**. Створивши кортеж, ви не можете змінити його елементи, додати нові або видалити існуючі.

Створення кортежів

Кортежі створюються за допомогою круглих дужок ():

```
# Порожній кортеж
```

```
empty_tuple = ()
```

```
print(empty_tuple) # ()
```

```
# Кортеж з числами
```

```
numbers = (1, 2, 3, 4, 5)
```

```
print(numbers) # (1, 2, 3, 4, 5)
```

```
# Кортеж з рядками
```

```
colors = ("червоний", "зелений", "синій")
```

```
print(colors) # ('червоний', 'зелений', 'синій')
```

```
# Змішаний кортеж
```

```
mixed = (42, "hello", 3.14, True)
```

```
print(mixed) # (42, 'hello', 3.14, True)
```

```
# Кортеж з одного елемента – зверніть увагу на кому!
```

```
single = (42,) # кома обов'язкова!  
not_a_tuple = (42) # це просто число 42, а не кортеж  
print(type(single)) # <class 'tuple'>  
print(type(not_a_tuple)) # <class 'int'>
```

Створення кортежу без дужок

У Python можна створити кортеж і без круглих дужок, просто перерахувавши значення через кому:

```
coordinates = 10, 20, 30  
print(coordinates) # (10, 20, 30)  
print(type(coordinates)) # <class 'tuple'>
```

Це часто використовується для повернення кількох значень з функції

```
def get_min_max(numbers):  
    return min(numbers), max(numbers) # повертає кортеж
```

```
result = get_min_max([5, 2, 8, 1, 9])  
print(result) # (1, 9)
```

Функція tuple()

Як і для списків, можна створити кортеж за допомогою вбудованої функції tuple():

```
# З рядка  
chars = tuple("Python")  
print(chars) # ('P', 'y', 't', 'h', 'o', 'n')
```

```
# Зі списку  
numbers = tuple([1, 2, 3, 4, 5])  
print(numbers) # (1, 2, 3, 4, 5)
```

```
# З діапазону  
range_tuple = tuple(range(1, 6))  
print(range_tuple) # (1, 2, 3, 4, 5)
```

Індексація та зрізи

До елементів кортежу можна звертатися за індексом, робити зрізи – все, як зі списками:

```
colors = ("червоний", "зелений", "синій", "жовтий", "фіолетовий")
```

```
# Доступ за індексом
```

```
print(colors[0]) # червоний
```

```
print(colors[-1]) # фіолетовий
```

```
# Зрізи
```

```
print(colors[1:4]) # ('зелений', 'синій', 'жовтий')
```

```
print(colors[::-1]) # ('фіолетовий', 'жовтий', 'синій', 'зелений', 'червоний')
```

```
# Довжина
```

```
print(len(colors)) # 5
```

```
# Перевірка наявності
```

```
print("синій" in colors) # True
```

```
print("білий" not in colors) # True
```

Незмінність кортежів – головна відмінність

Це найважливіша властивість кортежів. Ви не можете змінити кортеж після створення:

```
colors = ("червоний", "зелений", "синій")
```

```
# Спроба змінити елемент
```

```
# colors[0] = "жовтий" # TypeError: 'tuple' object does not support item assignment
```

```
# Спроба видалити елемент
```

```
# del colors[1] # TypeError: 'tuple' object doesn't support item deletion
```

```
# Спроба додати елемент
```

```
# colors.append("жовтий") # AttributeError: 'tuple' object has no attribute 'append'
```

```
# Але якщо кортеж містить список, сам список можна змінювати!
```

```
mixed = (1, 2, [3, 4])
```

```
mixed[2].append(5)
```

`print(mixed)` *# (1, 2, [3, 4, 5]) – сам кортеж не змінився, але список всередині змінився!*

Чому використовувати кортежі, якщо є списки?

Незмінність – це не недолік, а перевага в певних ситуаціях:

1. **Безпека даних:** Якщо дані не повинні змінюватися, кортеж гарантує це.

2. **Ключі в словниках:** Кортежі можна використовувати як ключі в словниках, списки – ні.

3. **Продуктивність:** Кортежі трохи швидші за списки, оскільки Python оптимізує роботу з ними.

4. **Семантика коду:** Кортежі показують, що дані "заморожені" – це полегшує розуміння коду.

Кортеж як ключ словника

```
coordinates = {}
```

```
coordinates[(10, 20)] = "Точка А"
```

```
coordinates[(30, 40)] = "Точка В"
```

```
print(coordinates) # {(10, 20): 'Точка А', (30, 40): 'Точка В'}
```

Список не можна використовувати як ключ

```
# coordinates[[1, 2]] = "Помилка" # TypeError: unhashable type: 'list'
```

Методи кортежів

Кортежі мають лише два методи:

```
numbers = (1, 2, 3, 2, 4, 2, 5)
```

count(x) – кількість входжень

```
print(numbers.count(2)) # 3
```

index(x) – індекс першого входження

```
print(numbers.index(3)) # 2
```

```
print(numbers.index(2)) # 1 (перше входження)
```

Коли використовувати кортежі, а коли списки?

Ситуація	Краще використовувати
Дані не повинні змінюватися (константи)	Кортеж

Ситуація	Краще використовувати
Потрібні ключі для словника	Кортеж
Функція повертає кілька значень	Кортеж (часто неявно)
Дані будуть часто змінюватися	Список
Потрібно додавати/видаляти елементи	Список
Потрібні методи sort(), reverse(), append()	Список

Практичний приклад: координати точок

Програма для роботи з геометричними фігурами

Точки представлені як кортежі (x, y)

point_a = (10, 20)

point_b = (30, 40)

point_c = (50, 60)

Список точок (сам список можна змінювати, додаючи нові точки)

triangle = [point_a, point_b, point_c]

print("ТРИКУТНИК:")

for i, point in enumerate(triangle, 1):

print(f"Точка {i}: {point}")

Функція обчислення відстані між двома точками

import math

def distance(p1, p2):

"""Повертає відстань між двома точками"""

return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

Обчислюємо периметр

perimeter = 0

for i in range(len(triangle)):


```

p1 = triangle[i]
p2 = triangle[(i + 1) % len(triangle)] # наступна точка (з замиканням)
dist = distance(p1, p2)
perimeter += dist
print(f"Відстань між точкою {i+1} і точкою {(i+1) % len(triangle) + 1}:
{dist:.2f}")

print(f"Периметр трикутника: {perimeter:.2f}")

# Спроба змінити координати точки
# point_a[0] = 100 # ПОМИЛКА! Кортеж незмінний

# Але ми можемо замінити всю точку в списку
triangle[0] = (100, 200)
print(f"Після заміни першої точки: {triangle}")

```

4. Вкладені списки та розпакування

Цей пункт об'єднує дві важливі теми: створення складніших структур даних (списки списків) та елегантний механізм розпакування, який робить код чистішим і зрозумілішим.

Вкладені списки (nested lists)

Списки можуть містити інші списки — це називається вкладеними списками. Вони ідеально підходять для представлення матриць, таблиць, дерев та інших складних структур.

```

# Проста матриця 3×3
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

```

```

print("Матриця:")
for row in matrix:
    for element in row:

```

```
print(element, end=" ")
print()
```

Доступ до елементів за двома індексами

```
print(f'Елемент [1][2]: {matrix[1][2]}') # 6 (рядок 1, стовпчик 2)
print(f'Елемент [2][0]: {matrix[2][0]}') # 7
```

Зміна елемента

```
matrix[0][1] = 99
print("Після зміни:")
for row in matrix:
    print(row)
```

Створення вкладених списків

Важливо правильно створювати вкладені списки, особливо коли вони великі:

Правильний спосіб створити матрицю 3×3 заповнену нулями

```
rows, cols = 3, 3
matrix = [[0 for _ in range(cols)] for _ in range(rows)]
print(matrix) # [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

НЕПРАВИЛЬНИЙ спосіб (не робіть так!)

```
wrong_matrix = [[0] * cols] * rows
print(wrong_matrix) # [[0, 0, 0], [0, 0, 0], [0, 0, 0]] – виглядає правильно,
але...
```

Але це пастка! Всі рядки посилаються на один і той самий список

```
wrong_matrix[0][0] = 99
print(wrong_matrix) # [[99, 0, 0], [99, 0, 0], [99, 0, 0]] – змінилося все!
```

Практичний приклад: таблиця множення

Створюємо таблицю множення 10×10

```
size = 10
multiplication_table = []

for i in range(1, size + 1):
```

```

row = []
for j in range(1, size + 1):
    row.append(i * j)
multiplication_table.append(row)

# Виводимо таблицю красиво
print("ТАБЛИЦЯ МНОЖЕННЯ")
print(" " + " ".join([f"{j:4}" for j in range(1, size + 1)]))
print(" " + "-" * (size * 5))

for i, row in enumerate(multiplication_table, 1):
    print(f"{i:2} |", end=" ")
    for val in row:
        print(f"{val:4}", end=" ")
    print()

```

Розпакування (unpacking)

Розпакування – це одна з найелегантніших особливостей Python, яка дозволяє присвоювати елементи послідовності окремим змінним.

```

# Просте розпакування кортежу
point = (10, 20)
x, y = point
print(f"x = {x}, y = {y}") # x = 10, y = 20

```

```

# Розпакування списку
coordinates = [100, 200, 300]
a, b, c = coordinates
print(f"a = {a}, b = {b}, c = {c}")

```

```

# Розпакування рядка
char1, char2, char3 = "ABC"
print(char1, char2, char3) # A B C

```

Розпакування зі зірочкою (*)

Коли кількість елементів невідома або ми хочемо зібрати частину з них, використовуємо *:

Збираємо "зайві" елементи в список

```
numbers = [1, 2, 3, 4, 5]
first, second, *rest = numbers
print(f"first = {first}") # 1
print(f"second = {second}") # 2
print(f"rest = {rest}") # [3, 4, 5]
```

Зірочка може бути не тільки в кінці

```
first, *middle, last = numbers
print(f"first = {first}") # 1
print(f"middle = {middle}") # [2, 3, 4]
print(f"last = {last}") # 5
```

Навіть на початку

```
*beginning, last = numbers
print(f"beginning = {beginning}") # [1, 2, 3, 4]
print(f"last = {last}") # 5
```

Розпакування в циклі

Особливо корисно при обході списків списків:

Список студентів (ім'я, вік, місто)

```
students = [
    ["Олена", 19, "Київ"],
    ["Іван", 20, "Львів"],
    ["Марія", 18, "Харків"],
    ["Петро", 21, "Одеса"]
]
```

Звичайний спосіб

```
for student in students:
    print(f"Ім'я: {student[0]}, Вік: {student[1]}, Місто: {student[2]}")
```

Краще з розпакуванням

```
for name, age, city in students:
    print(f"Ім'я: {name}, Вік: {age}, Місто: {city}")
```

Якщо потрібен індекс

```
for i, (name, age, city) in enumerate(students, 1):  
    print(f"{i}. {name}, {age} років, {city}")
```

Розпакування словників

Розпакування ключів та значень

```
person = {"name": "Олена", "age": 19, "city": "Київ"}
```

За замовчуванням ітеруємо по ключах

```
for key in person:  
    print(f"{key}: {person[key]}")
```

Краще з items()

```
for key, value in person.items():  
    print(f"{key}: {value}")
```

Практичний приклад: обробка даних з розпакуванням

Програма для аналізу результатів змагань

```
results = [  
    ("Іваненко", 9.85, 8.92, 9.33),  
    ("Петренко", 8.95, 9.12, 8.78),  
    ("Сидоренко", 9.22, 8.67, 9.41),  
    ("Коваленко", 8.88, 9.05, 8.99)  
]
```

```
print("🏆 РЕЗУЛЬТАТИ ЗМАГАНЬ")
```

```
print("=" * 60)
```

```
print(f"{'Прізвище':<12} {'Спроба 1':<10} {'Спроба 2':<10} {'Спроба  
3':<10} {'Середнє':<10}")
```

```
print("-" * 60)
```

```
for surname, *scores in results:
```

```
    average = sum(scores) / len(scores)
```

```
    print(f"{surname:<12} {scores[0]:<10.2f} {scores[1]:<10.2f}  
{scores[2]:<10.2f} {average:<10.2f}")
```

Знаходимо переможця

```

best_athlete = ""
best_average = 0

for surname, *scores in results:
    average = sum(scores) / len(scores)
    if average > best_average:
        best_average = average
        best_athlete = surname

print("=" * 60)
print(f"🏆 ПЕРЕМОЖЕЦЬ: {best_athlete} з середнім балом {best_average:.2f}")

```

Об'єднання списків (zip)

Функція `zip()` дозволяє об'єднувати кілька списків в один, групуючи елементи за індексами:

```

names = ["Олена", "Іван", "Марія", "Петро"]
ages = [19, 20, 18, 21]
cities = ["Київ", "Львів", "Харків", "Одеса"]

# Об'єднуємо списки
for name, age, city in zip(names, ages, cities):
    print(f"{name}, {age} років, {city}")

# Створюємо список кортежів
students = list(zip(names, ages, cities))
print(students) # [('Олена', 19, 'Київ'), ('Іван', 20, 'Львів'), ...]

# Розпаковуємо назад
unzipped_names, unzipped_ages, unzipped_cities = zip(*students)
print(unzipped_names) # ('Олена', 'Іван', 'Марія', 'Петро')
print(unzipped_ages) # (19, 20, 18, 21)

```

Практичний приклад: створення бази даних студентів

Повноцінний приклад з використанням всіх вивчених концепцій

```

print("📖 БАЗА ДАНИХ СТУДЕНТІВ")

```

```
print("=" * 60)
```

Структура: список студентів, кожен студент – [id, ім'я, вік, оцінки]

```
students = [  
    [1, "Олена Петренко", 19, [85, 90, 78, 92]],  
    [2, "Іван Коваленко", 20, [75, 88, 95, 70]],  
    [3, "Марія Шевченко", 18, [95, 92, 88, 96]],  
    [4, "Петро Бондаренко", 21, [60, 75, 80, 65]],  
    [5, "Анна Кравченко", 20, [100, 98, 95, 99]]  
]
```

```
def print_student(student):
```

```
    """Виводить інформацію про одного студента з розпакуванням"""
```

```
    student_id, name, age, grades = student
```

```
    avg_grade = sum(grades) / len(grades)
```

```
    print(f"ID: {student_id} | {name} | Вік: {age} | Оцінки: {grades} | Сер.:  
{avg_grade:.2f}")
```

```
def print_all_students():
```

```
    """Виводить всіх студентів"""
```

```
    print("\n📁 СПИСОК СТУДЕНТІВ:")
```

```
    print("-" * 80)
```

```
    for student in students:
```

```
        print_student(student)
```

```
    print("-" * 80)
```

```
def add_student():
```

```
    """Додає нового студента"""
```

```
    print("\n➕ ДОДАВАННЯ НОВОГО СТУДЕНТА")
```

```
    name = input("Прізвище та ім'я: ").strip().title()
```

```
    age = int(input("Вік: "))
```

Генеруємо новий ID

```
    new_id = max(s[0] for s in students) + 1
```

Додаємо з порожнім списком оцінок

```
    students.append([new_id, name, age, []])
```

```

print(f"✓ Студента {name} додано з ID {new_id}")

def add_grade():
    """Додає оцінку студенту"""
    print("\n✎ ДОДАВАННЯ ОЦІНКИ")
    student_id = int(input("ID студента: "))

    # Шукаємо студента
    for student in students:
        if student[0] == student_id:
            _, name, _, grades = student # розпаковуємо, ігноруючи деякі поля
            grade = int(input(f"Оцінка для {name}: "))
            if 0 <= grade <= 100:
                grades.append(grade)
                print(f"✓ Оцінку {grade} додано")
            else:
                print("✗ Оцінка має бути від 0 до 100")
                break
        else:
            print(f"✗ Студента з ID {student_id} не знайдено")

def analyze_group():
    """Аналізує успішність групи"""
    print("\n📊 АНАЛІЗ ГРУПИ")

    # Збираємо всі оцінки
    all_grades = []
    for _, _, _, grades in students:
        all_grades.extend(grades)

    if not all_grades:
        print("✗ Немає оцінок для аналізу")
        return

    print(f"Загальна кількість оцінок: {len(all_grades)}")
    print(f"Середній бал групи: {sum(all_grades) / len(all_grades):.2f}")

```



```

print(f'Найвища оцінка: {max(all_grades)}')
print(f'Найнижча оцінка: {min(all_grades)}')

# Аналіз по студентах
print("\n📊 УСПІШНІСТЬ СТУДЕНТІВ:")
students_with_avg = []
for student in students:
    student_id, name, age, grades = student
    if grades:
        avg = sum(grades) / len(grades)
        students_with_avg.append((name, avg))
    else:
        students_with_avg.append((name, 0))

# Сортуємо за середнім балом
students_with_avg.sort(key=lambda x: x[1], reverse=True)

for rank, (name, avg) in enumerate(students_with_avg, 1):
    print(f'{rank}. {name}: {avg:.2f}')

def top_students():
    """Виводить топ-3 студентів"""
    print("\n🏆 ТОП-3 СТУДЕНТИ")

# Створюємо список кортежів (середній бал, ім'я)
ratings = []
for student in students:
    _, name, _, grades = student
    if grades:
        avg = sum(grades) / len(grades)
        ratings.append((avg, name))

# Сортуємо за спаданням
ratings.sort(reverse=True)

for i, (avg, name) in enumerate(ratings[:3], 1):
    print(f'{i}. {name} – {avg:.2f}')

```

Головне меню

while True:

```
print("\nМЕНЮ:")
print("1. Показати всіх студентів")
print("2. Додати студента")
print("3. Додати оцінку")
print("4. Аналіз групи")
print("5. Топ-3 студенти")
print("6. Вийти")
```

```
choice = input("Оберіть опцію (1-6): ").strip()
```

```
if choice == "1":
    print_all_students()
elif choice == "2":
    add_student()
elif choice == "3":
    add_grade()
elif choice == "4":
    analyze_group()
elif choice == "5":
    top_students()
elif choice == "6":
    print("👋 До побачення!")
    break
else:
    print("❌ Невірний вибір")
```

Цей приклад демонструє практично всі вивчені концепції:

- Створення та роботу з вкладеними списками
- Розпакування кортежів та списків у циклах
- Використання `_` для ігнорування непотрібних значень
- Сортуювання складних структур
- Функції для обробки даних

У наступній лекції ми перейдемо до словників та множин – ще потужніших структур даних, які дозволяють організовувати інформацію у вигляді пар "ключ-значення" та працювати з унікальними елементами.

Питання для обговорення

1. У чому основна відмінність між списком та кортежем? Чому в Python існують обидві структури, якщо вони такі схожі? Наведіть приклади, коли краще використовувати кортеж, а коли список.

2. Поясніть різницю між методами `append()` та `extend()`. Що буде результатом виконання наступного коду?

```
list1 = [1, 2, 3]
list1.append([4, 5])
list2 = [1, 2, 3]
list2.extend([4, 5])
print(list1, list2)
```

3. Як створити копію списку? Чому просте присвоєння `new_list = old_list` не створює копію? Як це впливає на зміну елементів у новому списку?

4. Що таке зрізи списків? Як за допомогою зрізу отримати список у зворотному порядку? Як отримати кожен другий елемент списку?

5. Поясніть різницю між методами `remove()`, `pop()` та оператором `del`. У яких ситуаціях доцільно використовувати кожен з них?

6. Що таке розпакування (unpacking) списків та кортежів? Наведіть приклад, де одночасно розпаковуються перший, останній та всі інші елементи списку.

7. Проаналізуйте код:

```
a = [1, 2, 3]
b = a
b.append(4)
print(a)
```

Чому змінився список `a`? Як уникнути цього ефекту?

8. Напишіть програму, яка приймає список чисел і повертає новий список, що містить тільки унікальні елементи (без використання `set()`). Як би ви розв'язали цю задачу з використанням множини?

9. Як відсортувати список у зворотному порядку? Поясніть різницю між методом `sort()` та функцією `sorted()`. Який з них змінює оригінальний список?

10. Чому кортежі можуть бути ключами в словнику, а списки – ні? Яка фундаментальна властивість кортежів це дозволяє?

ЛЕКЦІЯ 8

Словники та множини

План лекції

1. Словники як колекції ключ-значення: створення, доступ та базові операції.

Почнемо з основ: що таке словник, як його створити, як додавати, отримувати та змінювати елементи. Пояснимо, чому ключі мають бути унікальними та незмінними, і чому словники – це найшвидший спосіб пошуку даних.

2. Методи словників: ефективна робота з даними.

Детально розглянемо всі основні методи: `keys()`, `values()`, `items()` для отримання різних представлень; `get()` для безпечного доступу; `update()` для об'єднання словників; `pop()` та `popitem()` для видалення. Покажемо, як правильно ітерувати по словниках.

3. Множини: унікальність та операції над множинами.

Познайомимось з множинами – структурами, які зберігають лише унікальні елементи. Розберемо математичні операції над множинами: об'єднання, перетин, різницю. Покажемо, як множини допомагають ефективно вирішувати задачі на унікальність.

4. Практичне застосування: словники як бази даних, множини для фільтрації.

Об'єднаємо всі знання в практичних прикладах: створимо телефонний довідник, систему підрахунку голосів, програму для пошуку спільних елементів. Покажемо, як вибрати правильну структуру даних для конкретної задачі.

1. Словники як колекції ключ-значення: створення, доступ та базові операції

Уявіть, що вам потрібно зберігати інформацію про людину: ім'я, вік, місто, професію. Зі списками ми могли б зробити так: `person = ["Олена", 19, "Київ", "студент"]`. Але як потім згадати, що `person[1]` – це вік, а `person[2]` – місто? Це незручно і нечитабельно. Нам потрібна структура, де в кожного значення є своє ім'я (ключ). Саме для цього існують **словники (dictionaries)**.

Що таке словник?

Словник – це невпорядкована, змінювана колекція пар **ключ-значення**. Кожен ключ у словнику є унікальним і незмінним. Значення можуть бути будь-якого типу і можуть змінюватися.

У реальному житті словник працює так само, як звичайний словник: ви шукаєте слово (ключ) і отримуєте його визначення (значення).

Створення словників

Словники створюються за допомогою фігурних дужок {} або функції dict():

Порожній словник

```
empty_dict = {}  
print(empty_dict) # {}
```

Словник з даними

```
person = {  
    "name": "Олена",  
    "age": 19,  
    "city": "Київ",  
    "is_student": True  
}  
print(person) # {'name': 'Олена', 'age': 19, 'city': 'Київ', 'is_student': True}
```

За допомогою функції dict()

```
person2 = dict(name="Іван", age=20, city="Львів")  
print(person2) # {'name': 'Іван', 'age': 20, 'city': 'Львів'}
```

Зі списку пар

```
pairs = [("name", "Марія"), ("age", 18), ("city", "Харків")]  
person3 = dict(pairs)  
print(person3) # {'name': 'Марія', 'age': 18, 'city': 'Харків'}
```

Ключі можуть бути різних типів, але мають бути незмінними

Ключі – рядки (найпоширеніший випадок)

```
d1 = {"name": "Олена", "age": 19}
```

Ключі – числа

```
d2 = {1: "один", 2: "два", 3: "три"}  
print(d2[2]) # два
```

Ключі – кортежі (можна, бо вони незмінні)

```
d3 = {(10, 20): "точка А", (30, 40): "точка В"}
```

Ключі – списки НЕ МОЖНА (бо списки змінні)

```
# d4 = {[1,2]: "помилка"} # TypeError: unhashable type: 'list'
```

Доступ до елементів словника

Головна перевага словника – доступ за ключем, а не за індексом:

```
person = {  
    "name": "Олена",  
    "age": 19,  
    "city": "Київ",  
    "is_student": True  
}
```

Доступ за ключем

```
print(person["name"]) # Олена
```

```
print(person["age"]) # 19
```

Якщо ключа немає – отримуємо помилку

```
# print(person["phone"]) # KeyError: 'phone'
```

Безпечний доступ через get() (повертає None, якщо ключа немає)

```
print(person.get("name")) # Олена
```

```
print(person.get("phone")) # None
```

```
print(person.get("phone", "Немає")) # Немає (значення за замовчуванням)
```

Додавання та зміна елементів

Словники змінні, тому ми можемо додавати нові пари або змінювати існуючі:

```
person = {"name": "Олена", "age": 19}
```

Додавання нового ключа

```
person["city"] = "Київ"
```

```
person["phone"] = "+380501234567"
```

```
print(person)    # {'name': 'Олена', 'age': 19, 'city': 'Київ', 'phone':  
'+380501234567'}
```

Зміна існуючого значення

```
person["age"] = 20  
print(person)    # {'name': 'Олена', 'age': 20, 'city': 'Київ', 'phone':  
'+380501234567'}
```

Якщо ключ існує – оновлюємо, якщо ні – додаємо

```
person["email"] = "olena@example.com" # додаємо  
person["age"] = 21 # оновлюємо
```

Видалення елементів

```
person = {  
    "name": "Олена",  
    "age": 19,  
    "city": "Київ",  
    "phone": "+380501234567",  
    "email": "olena@example.com"  
}
```

del – видаляє пару за ключем

```
del person["phone"]  
print(person)    # {'name': 'Олена', 'age': 19, 'city': 'Київ', 'email':  
'olena@example.com'}
```

pop() – видаляє і повертає значення

```
email = person.pop("email")  
print(f"Видалено email: {email}")  
print(person)    # {'name': 'Олена', 'age': 19, 'city': 'Київ'}
```

popitem() – видаляє і повертає останню додану пару (як кортеж)

```
last_item = person.popitem()  
print(f"Видалено останній елемент: {last_item}")  
print(person)    # {'name': 'Олена', 'age': 19}
```

clear() – видаляє всі елементи

```
person.clear()
print(person) # {}
```

Перевірка наявності ключа

Найчастіше використовується оператор `in`:

```
person = {"name": "Олена", "age": 19, "city": "Київ"}
```

Перевірка наявності ключа

```
if "name" in person:
    print(f"Ім'я: {person['name']}")
```

```
if "phone" not in person:
    print("Телефон не вказано")
```

Перевірка значення (рідше)

```
if "Олена" in person.values():
    print("Олена є в словнику як значення")
```

Чому словники такі швидкі?

Словники в Python реалізовані через **хеш-таблиці**. Це означає, що пошук елемента за ключем відбувається миттєво, незалежно від розміру словника. У списку, щоб знайти елемент, нам потрібно перебрати всі елементи ($O(n)$). У словнику пошук займає практично константний час ($O(1)$).

Порівняння швидкості пошуку

```
import time
```

Створюємо великий список і словник

```
size = 1000000
data_list = list(range(size))
data_dict = {i: i for i in range(size)}
```

Шукаємо останній елемент

```
start = time.time()
print(999999 in data_list)
end = time.time()
print(f"Пошук у списку: {end - start:.6f} сек")
```



```
start = time.time()
```

```
print(999999 in data_dict)
```

```
end = time.time()
```

```
print(f"Пошук у словнику: {end - start:.6f} сек")
```

Різниця буде вражаючою! Словник знайде елемент майже миттєво, а списку доведеться перебрати мільйон елементів.

Практичний приклад: простий телефонний довідник

```
# Телефонний довідник
```

```
phonebook = {}
```

```
print("☎ ТЕЛЕФОННИЙ ДОВІДНИК")
```

```
print("=" * 40)
```

```
while True:
```

```
    print("\n1. Додати контакт")
```

```
    print("2. Знайти контакт")
```

```
    print("3. Показати всі контакти")
```

```
    print("4. Видалити контакт")
```

```
    print("5. Вийти")
```

```
choice = input("Оберіть опцію (1-5): ").strip()
```

```
if choice == "1":
```

```
    name = input("Ім'я: ").strip().title()
```

```
    phone = input("Телефон: ").strip()
```

```
    phonebook[name] = phone
```

```
    print(f"✓ Контакт {name} додано")
```

```
elif choice == "2":
```

```
    name = input("Введіть ім'я для пошуку: ").strip().title()
```

```
    phone = phonebook.get(name)
```

```
    if phone:
```

```
        print(f"📞 {name}: {phone}")
```

```
    else:
```

```

    print(f"✗ Контакт {name} не знайдено")

elif choice == "3":
    if not phonebook:
        print("📖 Довідник порожній")
    else:
        print("\n📁 СПИСОК КОНТАКТІВ:")
        for name, phone in phonebook.items():
            print(f" {name}: {phone}")

elif choice == "4":
    name = input("Введіть ім'я для видалення: ").strip().title()
    if name in phonebook:
        del phonebook[name]
        print(f"✓ Контакт {name} видалено")
    else:
        print(f"✗ Контакт {name} не знайдено")

elif choice == "5":
    print("До побачення!")
    break

else:
    print("✗ Невірний вибір")

```

2. Методи словників: ефективна робота з даними

Словники мають багатий набір методів, які роблять роботу з ними ефективною та зручною.

Отримання всіх ключів, значень та пар

```

student = {
    "name": "Марія Шевченко",
    "age": 19,
    "course": 2,

```

```

    "faculty": "Комп'ютерні науки",
    "average_grade": 92.5
}

# keys() – повертає всі ключі
print(student.keys())      # dict_keys(['name', 'age', 'course', 'faculty',
                             'average_grade'])
# Можна ітерувати
for key in student.keys():
    print(f"Ключ: {key}")

# values() – повертає всі значення
print(student.values())    # dict_values(['Марія Шевченко', 19, 2,
    'Комп\'ютерні науки', 92.5])
for value in student.values():
    print(f"Значення: {value}")

# items() – повертає всі пари як кортежі (ключ, значення)
print(student.items())    # dict_items([('name', 'Марія Шевченко'), ('age', 19),
...])
for key, value in student.items():
    print(f"{key}: {value}")

```

Метод get() – безпечний доступ

Ми вже бачили get(), але він має ще одну важливу особливість – значення за замовчуванням:

```
student = {"name": "Марія", "age": 19}
```

```

# Звичайний доступ – ризик помилки
# print(student["course"]) # KeyError!

```

```

# Безпечний доступ
print(student.get("course")) # None
print(student.get("course", "Немає")) # Немає

```

```
# Це особливо корисно при підрахунках
```

```
words = ["яблуко", "груша", "яблуко", "слива", "яблуко", "груша"]
```

```
counter = {}
```

```
for word in words:
```

```
    # Безпечно отримуємо поточне значення, якщо ключа немає –  
    починаємо з 0
```

```
    counter[word] = counter.get(word, 0) + 1
```

```
print(counter) # {'яблуко': 3, 'груша': 2, 'слива': 1}
```

Метод `setdefault()` – отримати або створити

Метод `setdefault()` повертає значення за ключем, а якщо ключа немає – створює нову пару зі значенням за замовчуванням:

```
student = {"name": "Марія"}
```

```
# Якщо ключ є – повертає значення
```

```
age = student.setdefault("age", 18)
```

```
print(age) # 18 (бо ключа не було, створив і повернув 18)
```

```
print(student) # {'name': 'Марія', 'age': 18}
```

```
# Якщо ключ є – повертає існуюче значення
```

```
age = student.setdefault("age", 20)
```

```
print(age) # 18 (існуюче значення, не змінилося!)
```

```
print(student) # {'name': 'Марія', 'age': 18}
```

Метод `update()` – об'єднання словників

```
student = {"name": "Марія", "age": 19}
```

```
additional_info = {"city": "Львів", "faculty": "Комп'ютерні науки", "age": 20}
```

```
# update() додає пари з іншого словника, оновлюючи існуючі
```

```
student.update(additional_info)
```

```
print(student) # {'name': 'Марія', 'age': 20, 'city': 'Львів', 'faculty':  
'Комп\ютерні науки'}
```

```
# Можна оновлювати зі списку пар
```

```
student.update([("course", 2), ("group", "KH-21")])
```

```
print(student)
```

Або з іменованих аргументів

```
student.update(email="maria@example.com", phone="+380501234567")  
print(student)
```

Метод pop() та popitem() – видалення з поверненням

```
student = {  
    "name": "Марія",  
    "age": 19,  
    "city": "Львів",  
    "faculty": "Комп'ютерні науки",  
    "course": 2  
}
```

pop(key) – видаляє і повертає значення

```
age = student.pop("age")  
print(f"Видалено вік: {age}")  
print(student) # без ключа 'age'
```

pop(key, default) – безпечне видалення

```
phone = student.pop("phone", "немає")  
print(f"Телефон: {phone}") # немає (помилки не було)
```

popitem() – видаляє і повертає останню додану пару

(корисно, коли порядок важливий)

```
last = student.popitem()  
print(f"Видалено останній елемент: {last}")
```

Копіювання словників

Як і зі списками, просте присвоєння не створює копію:

```
original = {"name": "Марія", "grades": [90, 85, 92]}  
copy = original # це не копія!
```

```
copy["name"] = "Олена"  
print(original["name"]) # Олена – змінилося!
```

```

# copy() – поверхнева копія
original = {"name": "Марія", "grades": [90, 85, 92]}
copy = original.copy()

copy["name"] = "Олена"
print(original["name"]) # Марія – не змінилося! Добре.

# Але для вкладених структур це не працює
copy["grades"].append(100)
print(original["grades"]) # [90, 85, 92, 100] – змінилося!

# Для глибокого копіювання потрібен copy.deepcopy()
import copy
original = {"name": "Марія", "grades": [90, 85, 92]}
deep_copy = copy.deepcopy(original)

deep_copy["grades"].append(100)
print(original["grades"]) # [90, 85, 92] – не змінилося!

```

Ітерування по словниках

Існує кілька способів пройти по словнику:

```

student = {
    "name": "Марія Шевченко",
    "age": 19,
    "course": 2,
    "faculty": "Комп'ютерні науки"
}

```

```

# 1. По ключах (найчастіше)
for key in student:
    print(f"{key}: {student[key]}")

```

```

# 2. По ключах явно (те саме)
for key in student.keys():
    print(f"{key}: {student[key]}")

```

```

# 3. По значеннях (рідше)

```

```
for value in student.values():
    print(value)
```

4. По парах ключ-значення (найкраще!)

```
for key, value in student.items():
    print(f"{key}: {value}")
```

Створення словників за допомогою dict comprehensions

Як і для списків, для словників є свої включення:

Квадрати чисел

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

З умовою

```
even_squares = {x: x**2 for x in range(1, 11) if x % 2 == 0}
print(even_squares) # {2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

З двох списків

```
names = ["Олена", "Іван", "Марія"]
ages = [19, 20, 18]
students = {name: age for name, age in zip(names, ages)}
print(students) # {'Олена': 19, 'Іван': 20, 'Марія': 18}
```

Інвертування словника (ключі стають значеннями і навпаки)

```
original = {"a": 1, "b": 2, "c": 3}
inverted = {value: key for key, value in original.items()}
print(inverted) # {1: 'a', 2: 'b', 3: 'c'}
```

Практичний приклад: підрахунок голосів на виборах

Програма для підрахунку голосів на виборах

```
votes = [
    "Петренко", "Коваленко", "Шевченко", "Петренко",
    "Коваленко", "Петренко", "Бондаренко", "Шевченко",
    "Петренко", "Коваленко", "Шевченко", "Петренко",
    "Коваленко", "Петренко", "Петренко"
]
```

```
print("🗳️ ПІДРАХУНОК ГОЛОСІВ")
print("=" * 40)
```

Спосіб 1: звичайний підрахунок з get()

```
results = {}
```

```
for candidate in votes:
```

```
    results[candidate] = results.get(candidate, 0) + 1
```

```
print("\nРезультати голосування:")
```

```
for candidate, count in sorted(results.items(), key=lambda x: x[1],
reverse=True):
```

```
    print(f" {candidate}: {count} голосів ({count/len(votes)*100:.1f}%)")
```

Спосіб 2: з використанням Counter (з collections)

```
from collections import Counter
```

```
counter = Counter(votes)
```

```
print("\nТоп-3 кандидати:")
```

```
for candidate, count in counter.most_common(3):
```

```
    print(f" {candidate}: {count} голосів")
```

Знаходимо переможця

```
winner, votes_count = counter.most_common(1)[0]
```

```
print(f"\n🏆 ПЕРЕМОЖЕЦЬ: {winner} з {votes_count} голосами")
```

Аналіз за алфавітом

```
print("\n📁 Кандидати за алфавітом:")
```

```
for candidate in sorted(results.keys()):
```

```
    print(f" {candidate}: {results[candidate]}")
```

Практичний приклад: словник як кеш (memoізація)

Демонстрація використання словника для кешування результатів

```
def fibonacci_with_cache(n, cache={}):
```

```
    """Обчислює число Фібоначчі з кешуванням результатів"""
```

```
    if n in cache:
```



```

    return cache[n]

    if n <= 1:
        result = n
    else:
        result = fibonacci_with_cache(n-1, cache) + fibonacci_with_cache(n-2,
cache)

    cache[n] = result
    return result

# Обчислюємо числа Фібоначчі
import time

n = 40

# Без кешу (звичайна рекурсія) – дуже повільно!
def fib_slow(n):
    if n <= 1:
        return n
    return fib_slow(n-1) + fib_slow(n-2)

start = time.time()
# result = fib_slow(n) # для n=40 це може зайняти багато часу!
end = time.time()
# print(f"Без кешу: {end - start:.2f} сек")

# З кешем – миттєво!
start = time.time()
result = fibonacci_with_cache(n)
end = time.time()
print(f"З кешем: {result} (обчислено за {end - start:.6f} сек)")

# Повторний виклик – ще швидше (бере з кеша)
start = time.time()
result = fibonacci_with_cache(n)
end = time.time()

```

```
print(f"З кешем (повторно): {result} (обчислено за {end - start:.6f} сек)")
```

```
# Подивимося на вміст кеша
```

```
print(f"\nКеш містить {len(fibonacci_with_cache.__defaults__[0])} значень")
```

```
print("Перші 10 значень з кеша:")
```

```
for i in range(10):
```

```
    print(f" fib({i}) = {fibonacci_with_cache.__defaults__[0][i]}")
```

Практичний приклад: групування даних

```
# Групування студентів за курсами
```

```
students = [
```

```
    ("Олена Петренко", 2, 92),
```

```
    ("Іван Коваленко", 1, 78),
```

```
    ("Марія Шевченко", 3, 95),
```

```
    ("Петро Бондаренко", 2, 85),
```

```
    ("Анна Кравченко", 1, 88),
```

```
    ("Дмитро Мельник", 3, 91),
```

```
    ("Катерина Лисенко", 2, 79),
```

```
    ("Олександр Гриценко", 4, 82)
```

```
]
```

```
# Групуємо за курсом
```

```
grouped = {}
```

```
for name, course, grade in students:
```

```
    # Якщо курсу ще немає в словнику, створюємо список
```

```
    if course not in grouped:
```

```
        grouped[course] = []
```

```
    grouped[course].append((name, grade))
```

```
# Виводимо результат
```

```
print("📋 ГРУПУВАННЯ СТУДЕНТІВ ЗА КУРСАМИ")
```

```
print("=" * 60)
```

```
for course in sorted(grouped.keys()):
```

```
    students_list = grouped[course]
```

```
    print(f"\nКурс {course}:")
```

```
for name, grade in students_list:
    print(f" {name}: {grade}")
```

```
# Статистика по курсу
```

```
grades = [g for _, g in students_list]
avg_grade = sum(grades) / len(grades)
print(f" → Середній бал: {avg_grade:.2f}")
print(f" → Кількість студентів: {len(students_list)}")
```

У наступних пунктах ми розглянемо множини – ще одну потужну структуру даних для роботи з унікальними елементами та операціями над множинами.

3. Множини: унікальність та операції над множинами

Множини (sets) – це ще один вбудований тип даних у Python. Вони дуже схожі на математичні множини, які ви вивчали в школі. Множина – це **невпорядкована колекція унікальних елементів**. Головні особливості множин:

- Всі елементи унікальні (дублікати автоматично видаляються)
- Елементи неупорядковані (не можна звернутися за індексом)
- Елементи мають бути незмінними (як ключі словників)

Створення множин

Множини створюються за допомогою фігурних дужок {} або функції set():

```
# Порожня множина (увага: не {}! це порожній словник)
```

```
empty_set = set()
print(empty_set) # set()
print(type(empty_set)) # <class 'set'>
```

```
# Порожній словник – це {}, а не множина
```

```
empty_dict = {}
print(type(empty_dict)) # <class 'dict'>
```

```
# Множина з даними
```

```
fruits = {"яблуко", "банан", "апельсин", "груша"}
```

`print(fruits)` # {'яблуко', 'груша', 'банан', 'апельсин'} (порядок може бути різним)

З дублікатами – вони автоматично видаляються

`numbers = {1, 2, 3, 2, 1, 4, 3, 5}`

`print(numbers)` # {1, 2, 3, 4, 5}

За допомогою set() з інших послідовностей

`chars = set("hello world")`

`print(chars)` # {' ', 'h', 'e', 'l', 'o', 'w', 'r', 'd'} – літери унікальні

`words = set(["яблуко", "банан", "яблуко", "апельсин", "банан"])`

`print(words)` # {'апельсин', 'яблуко', 'банан'}

З діапазону

`numbers = set(range(1, 10))`

`print(numbers)` # {1, 2, 3, 4, 5, 6, 7, 8, 9}

Основи роботи з множинами

`fruits = {"яблуко", "банан", "апельсин"}`

Додавання елемента

`fruits.add("груша")`

`print(fruits)` # {'яблуко', 'груша', 'банан', 'апельсин'}

Додавання існуючого елемента – нічого не змінюється

`fruits.add("яблуко")`

`print(fruits)` # {'яблуко', 'груша', 'банан', 'апельсин'} (без змін)

Видалення елемента

`fruits.remove("банан")` # якщо елемента немає – помилка

`print(fruits)` # {'яблуко', 'груша', 'апельсин'}

Безпечне видалення (не генерує помилку)

`fruits.discard("ківі")` # просто нічого не робить

`print(fruits)`

Видалення випадкового елемента

```
popped = fruits.pop() # видаляє і повертає довільний елемент
print(f'Видалено: {popped}')
print(fruits)
```

Очищення множини

```
fruits.clear()
print(fruits) # set()
```

Перевірка належності

Як і для словників, перевірка наявності елемента в множині відбувається дуже швидко ($O(1)$):

```
fruits = {"яблуко", "банан", "апельсин", "груша"}
```

Перевірка належності

```
print("банан" in fruits) # True
print("ківі" not in fruits) # True
```

Це набагато швидше, ніж для списків!

```
import time
```

```
size = 1000000
```

```
data_list = list(range(size))
```

```
data_set = set(range(size))
```

```
start = time.time()
```

```
print(999999 in data_list)
```

```
end = time.time()
```

```
print(f'Пошук у списку: {end - start:.6f} сек')
```

```
start = time.time()
```

```
print(999999 in data_set)
```

```
end = time.time()
```

```
print(f'Пошук у множині: {end - start:.6f} сек')
```

Операції над множинами

Це найпотужніша особливість множин. Python підтримує всі класичні математичні операції над множинами.

Створюємо дві множини

```
set_a = {1, 2, 3, 4, 5}
```

```
set_b = {4, 5, 6, 7, 8}
```

```
print(f"Множина A: {set_a}")
```

```
print(f"Множина B: {set_b}")
```

```
print()
```

1. Об'єднання (union) – всі елементи з обох множин

```
union1 = set_a | set_b
```

```
union2 = set_a.union(set_b)
```

```
print(f"A | B: {union1}") # {1, 2, 3, 4, 5, 6, 7, 8}
```

2. Перетин (intersection) – елементи, які є в обох множинах

```
intersection1 = set_a & set_b
```

```
intersection2 = set_a.intersection(set_b)
```

```
print(f"A & B: {intersection1}") # {4, 5}
```

3. Різниця (difference) – елементи, які є в A, але немає в B

```
difference1 = set_a - set_b
```

```
difference2 = set_a.difference(set_b)
```

```
print(f"A - B: {difference1}") # {1, 2, 3}
```

4. Симетрична різниця (symmetric difference) – елементи, які є в одній множині, але не в обох

```
sym_diff1 = set_a ^ set_b
```

```
sym_diff2 = set_a.symmetric_difference(set_b)
```

```
print(f"A ^ B: {sym_diff1}") # {1, 2, 3, 6, 7, 8}
```

Порівняння множин

Підмножина (subset)

```
set_c = {1, 2, 3}
```

```
set_d = {1, 2, 3, 4, 5}
```

```

print(set_c.issubset(set_d)) # True – всі елементи C є в D
print(set_c <= set_d)      # те саме

# Надмножина (superset)
print(set_d.issuperset(set_c)) # True – D містить всі елементи C
print(set_d >= set_c)        # те саме

# Рівність
set_e = {1, 2, 3}
set_f = {3, 2, 1}
print(set_e == set_f) # True – порядок не має значення

# Чи є множини неперетинними (disjoint)
set_g = {1, 2, 3}
set_h = {4, 5, 6}
print(set_g.isdisjoint(set_h)) # True – немає спільних елементів

```

Незмінні множини (frozenset)

Іноді потрібна множина, яку не можна змінювати – наприклад, для використання як ключа в словнику. Для цього існує frozenset:

```

# Звичайна множина – змінювана
normal_set = {1, 2, 3}
normal_set.add(4) # працює

# Frozenset – незмінна
frozen = frozenset([1, 2, 3, 4, 5])
# frozen.add(6) # AttributeError: 'frozenset' object has no attribute 'add'

# Frozenset можна використовувати як ключ у словнику
dictionary = {
    frozenset([1, 2, 3]): "множина А",
    frozenset([4, 5, 6]): "множина В"
}
print(dictionary[frozenset([1, 2, 3])]) # множина А

```

Практичний приклад: пошук спільних друзів у соціальній мережі

Соціальна мережа: друзі користувачів

```
users_friends = {  
    "Олена": {"Марія", "Іван", "Петро", "Анна"},  
    "Іван": {"Марія", "Олена", "Дмитро", "Катерина"},  
    "Марія": {"Олена", "Іван", "Анна", "Дмитро"},  
    "Петро": {"Олена", "Анна", "Дмитро"},  
    "Анна": {"Олена", "Марія", "Петро", "Катерина"}  
}
```

```
print("👤 АНАЛІЗ ДРУЗІВ У СОЦМЕРЕЖІ")
```

```
print("=" * 60)
```

```
def find_mutual_friends(user1, user2):
```

```
    """Знаходить спільних друзів двох користувачів"""
```

```
    if user1 not in users_friends or user2 not in users_friends:
```

```
        return set()
```

```
    return users_friends[user1] & users_friends[user2]
```

```
def suggest_friends(user):
```

```
    """Пропонує потенційних друзів (друзі друзів, які ще не є друзями)"""
```

```
    if user not in users_friends:
```

```
        return set()
```

```
    my_friends = users_friends[user]
```

```
    potential = set()
```

```
    for friend in my_friends:
```

```
        # Додаємо друзів мого друга
```

```
        potential |= users_friends.get(friend, set())
```

```
    # Видаляємо себе і вже існуючих друзів
```

```
    potential -= my_friends
```

```
    potential.discard(user)
```

```
    return potential
```

```
def friendship_degrees(user):
```



```

"""Аналізує ступені зв'язку (друзі, друзі друзів, тощо)"""
if user not in users_friends:
    return {}

result = {}
my_friends = users_friends[user]

# Перший ступінь
result[1] = my_friends

# Другий ступінь (друзі друзів)
second_degree = set()
for friend in my_friends:
    second_degree |= users_friends.get(friend, set())
second_degree -= my_friends
second_degree.discard(user)
result[2] = second_degree

return result

# Демонстрація
print("\nСпільні друзі Олени та Івана:")
mutual = find_mutual_friends("Олена", "Іван")
print(f" {mutual}")

print("\nПотенційні друзі для Анни:")
suggestions = suggest_friends("Анна")
print(f" {suggestions}")

print("\nАналіз зв'язків для Олени:")
degrees = friendship_degrees("Олена")
print(" Друзі (1 ступінь):", degrees[1])
print(" Друзі друзів (2 ступінь):", degrees[2])

# Статистика
print("\n📊 СТАТИСТИКА:")
all_users = set(users_friends.keys())

```

```

all_friendships = set()

for user, friends in users_friends.items():
    all_friendships |= friends

print(f'Всього користувачів: {len(all_users)}')
print(f'Всього унікальних осіб у друзях: {len(all_friendships)}')
print(f'Загальна кількість дружніх зв'язків: {sum(len(f) for f in
users_friends.values())}')

```

4. Практичне застосування: словники як бази даних, множини для фільтрації

Тепер, коли ми знаємо всі основні структури даних, настав час об'єднати їх у потужних практичних прикладах. Ми побачимо, як правильний вибір структури даних може зробити код ефективним та елегантним.

Приклад 1: Система управління бібліотекою

Комплексна система для управління бібліотекою

```

library = {
    "books": {
        "978-0-13-595705-9": {
            "title": "Python Crash Course",
            "author": "Eric Matthes",
            "year": 2019,
            "copies": 3,
            "available": 2
        },
        "978-1-49-205776-7": {
            "title": "Fluent Python",
            "author": "Luciano Ramalho",
            "year": 2022,
            "copies": 2,
            "available": 1
        },
        "978-0-596-15806-4": {

```

```

        "title": "Learning Python",
        "author": "Mark Lutz",
        "year": 2013,
        "copies": 1,
        "available": 0
    }
},
"users": {
    "user1": {"name": "Олена Петренко", "books": []},
    "user2": {"name": "Іван Коваленко", "books": ["978-0-13-595705-9"]},
    "user3": {"name": "Марія Шевченко", "books": []}
},
"genres": {
    "programming": {"978-0-13-595705-9", "978-1-49-205776-7", "978-0-596-15806-4"},
    "python": {"978-0-13-595705-9", "978-1-49-205776-7", "978-0-596-15806-4"},
    "tutorial": {"978-0-13-595705-9", "978-0-596-15806-4"}
}
}

```

```

print("📖 СИСТЕМА УПРАВЛІННЯ БІБЛІОТЕКОЮ")

```

```

print("=" * 70)

```

```

def search_books(query):

```

```

    """Пошук книг за автором або назвою"""

```

```

    results = []

```

```

    query = query.lower()

```

```

    for isbn, book in library["books"].items():

```

```

        if query in book["title"].lower() or query in book["author"].lower():

```

```

            results.append((isbn, book))

```

```

    return results

```

```

def borrow_book(user_id, isbn):

```

```

    """Видача книги користувачу"""

```

```

if user_id not in library["users"]:
    return "✗ Користувача не знайдено"

if isbn not in library["books"]:
    return "✗ Книгу не знайдено"

book = library["books"][isbn]
user = library["users"][user_id]

if book["available"] <= 0:
    return "✗ Книга недоступна"

if len(user["books"]) >= 3:
    return "✗ Ліміт книг (максимум 3)"

# Видаємо книгу
book["available"] -= 1
user["books"].append(isbn)
return f"✓ Книгу '{book['title']}' видано користувачу {user['name']}"

def return_book(user_id, isbn):
    """Повернення книги"""
    if user_id not in library["users"]:
        return "✗ Користувача не знайдено"

    user = library["users"][user_id]

    if isbn not in user["books"]:
        return "✗ У цього користувача немає такої книги"

    book = library["books"][isbn]
    book["available"] += 1
    user["books"].remove(isbn)

    return f"✓ Книгу '{book['title']}' повернено"

```

```

def get_recommendations(user_id):
    """Рекомендація книг на основі жанрів уже взятих книг"""
    if user_id not in library["users"]:
        return set()

    user_books = library["users"][user_id]["books"]
    if not user_books:
        return set()

    # Збираємо всі жанри, які цікавлять користувача
    user_genres = set()
    for isbn in user_books:
        for genre, books in library["genres"].items():
            if isbn in books:
                user_genres.add(genre)

    # Рекомендуємо книги з цих жанрів, яких ще немає у користувача
    recommendations = set()
    for genre in user_genres:
        recommendations |= library["genres"][genre]

    recommendations -= set(user_books)

    # Фільтруємо тільки доступні книги
    available = {isbn for isbn, book in library["books"].items()
                  if book["available"] > 0}

    return recommendations & available

# Демонстрація
print("\n🔍 Пошук книг за запитом 'python':")
for isbn, book in search_books("python"):
    print(f"ISBN: {book['title']} — {book['author']} ({book['available']}/{book['copies']})")

print("\n📖 Видача книги:")
result = borrow_book("user3", "978-0-13-595705-9")

```

```
print(result)
```

```
print("\n📖 Стан бібліотеки після видачі:")
```

```
for isbn, book in library["books"].items():
```

```
    print(f" {book['title']}: {book['available']}/{book['copies']}")
```

```
print("\n👤 Рекомендації для Івана (user2):")
```

```
recommended = get_recommendations("user2")
```

```
for isbn in recommended:
```

```
    book = library["books"][isbn]
```

```
    print(f" {book['title']} – {book['author']}")
```

Приклад 2: Аналіз логів сервера

```
import re
```

```
from collections import defaultdict, Counter
```

```
# Імітація лог-файлу сервера
```

```
log_lines = [
```

```
    "192.168.1.100 - - [10/Oct/2023:13:55:36 +0000] GET /index.html HTTP/1.1  
200 2326",
```

```
    "192.168.1.101 - - [10/Oct/2023:13:56:12 +0000] GET /about.html HTTP/1.1  
200 1542",
```

```
    "192.168.1.100 - - [10/Oct/2023:13:57:23 +0000] GET /style.css HTTP/1.1  
200 1245",
```

```
    "192.168.1.102 - - [10/Oct/2023:13:58:45 +0000] POST /login HTTP/1.1 302  
0",
```

```
    "192.168.1.100 - - [10/Oct/2023:13:59:01 +0000] GET /dashboard HTTP/1.1  
403 234",
```

```
    "192.168.1.103 - - [10/Oct/2023:14:01:32 +0000] GET /index.html HTTP/1.1  
200 2326",
```

```
    "192.168.1.101 - - [10/Oct/2023:14:02:54 +0000] GET /products.html  
HTTP/1.1 200 3421",
```

```
    "192.168.1.104 - - [10/Oct/2023:14:03:17 +0000] GET /index.html HTTP/1.1  
404 123",
```

```
    "192.168.1.100 - - [10/Oct/2023:14:05:43 +0000] GET /api/data HTTP/1.1  
200 567",
```

```

0",
    ]

```

```

print("\n📊 АНАЛІЗ ЛОГІВ СЕРВЕРА")

```

```

print("=" * 70)

```

```

# Парсимо логи

```

```

parsed_logs = []

```

```

pattern = r'(\d+\.\d+\.\d+\.\d+) .*? \[(.*?)\] "(GET|POST) (.*?) HTTP.*?" (\d+)
(\d+)'

```

```

for line in log_lines:

```

```

    match = re.search(pattern, line)

```

```

    if match:

```

```

        ip, timestamp, method, url, status, size = match.groups()

```

```

        parsed_logs.append({

```

```

            "ip": ip,

```

```

            "timestamp": timestamp,

```

```

            "method": method,

```

```

            "url": url,

```

```

            "status": int(status),

```

```

            "size": int(size)

```

```

        })

```

```

# Статистика за IP-адресами

```

```

ip_stats = defaultdict(lambda: {"count": 0, "methods": set(), "urls": set(),
"total_size": 0})

```

```

for log in parsed_logs:

```

```

    ip = log["ip"]

```

```

    ip_stats[ip]["count"] += 1

```

```

    ip_stats[ip]["methods"].add(log["method"])

```

```

    ip_stats[ip]["urls"].add(log["url"])

```

```

    ip_stats[ip]["total_size"] += log["size"]

```

```

print("\n🌐 СТАТИСТИКА ЗА IP-АДРЕСАМИ:")

```

```

for ip, stats in sorted(ip_stats.items(), key=lambda x: x[1]["count"],
reverse=True):
    print(f" {ip}:")
    print(f"   Запитів: {stats['count']}")
    print(f"   Методи: {', '.join(stats['methods'])}")
    print(f"   Унікальних URL: {len(stats['urls'])}")
    print(f"   Загальний трафік: {stats['total_size']} байт")

# Статистика за статус-кодами
status_counts = Counter(log["status"] for log in parsed_logs)

print("\n✅ СТАТИСТИКА ЗА СТАТУС-КОДАМИ:")
for status, count in sorted(status_counts.items()):
    print(f" {status}: {count} запитів")

# Найпопулярніші URL
url_counts = Counter(log["url"] for log in parsed_logs)

print("\n↔ НАЙПОПУЛЯРНІШІ URL:")
for url, count in url_counts.most_common(3):
    print(f" {url}: {count} запитів")

# Аналіз часових проміжків
hour_stats = defaultdict(lambda: {"count": 0, "unique_ips": set()})

for log in parsed_logs:
    # Витягуємо годину з timestamp
    hour = int(log["timestamp"].split(":")[1])
    hour_stats[hour]["count"] += 1
    hour_stats[hour]["unique_ips"].add(log["ip"])

print("\n□ СТАТИСТИКА ЗА ГОДИНАМИ:")
for hour in sorted(hour_stats.keys()):
    stats = hour_stats[hour]
    print(f" {hour:02d}:00 – {stats['count']} запитів, {len(stats['unique_ips'])} унікальних IP")

```



```

# Виявлення підозрілої активності
suspicious_ips = set()
for ip, stats in ip_stats.items():
    # Багато 404 помилок
    error_count = sum(1 for log in parsed_logs if log["ip"] == ip and log["status"]
    >= 400)
    if error_count > 2:
        suspicious_ips.add(ip)

# Доступ до закритих URL
restricted_access = any(log["url"] in ["/dashboard", "/admin", "/api/data"]
    and log["status"] == 403 for log in parsed_logs if log["ip"] ==
ip)
    if restricted_access:
        suspicious_ips.add(ip)

print("\n🚨 ПІДОЗРІЛА АКТИВНІСТЬ:")
if suspicious_ips:
    for ip in suspicious_ips:
        print(f"⚠️ {ip}")
else:
    print("✅ Підозрілої активності не виявлено")

```

Приклад 3: Система рекомендацій фільмів

```

# Система рекомендацій фільмів на основі вподобань користувачів
movies = {
    "matrix": {"title": "Матриця", "year": 1999, "genre": {"sci-fi", "action"}},
    "inception": {"title": "Початок", "year": 2010, "genre": {"sci-fi", "thriller"}},
    "interstellar": {"title": "Інтерстеллар", "year": 2014, "genre": {"sci-fi",
"drama"}},
    "godfather": {"title": "Хрещений батько", "year": 1972, "genre": {"drama",
"crime"}},
    "pulp_fiction": {"title": "Кримінальне чтиво", "year": 1994, "genre":
{"crime", "comedy"}},
    "avengers": {"title": "Месники", "year": 2012, "genre": {"action", "sci-fi"}},
    "titanic": {"title": "Титанік", "year": 1997, "genre": {"drama", "romance"}},

```

```

    "shining": {"title": "Сяйво", "year": 1980, "genre": {"horror", "thriller"}},
    "ring": {"title": "Дзвінок", "year": 2002, "genre": {"horror", "mystery"}},
    "forrest": {"title": "Форрест Гамп", "year": 1994, "genre": {"drama",
"comedy"}}}
    }

```

```

users = {
    "alice": {
        "liked": {"inception", "interstellar", "matrix"},
        "disliked": {"titanic", "ring"}
    },
    "bob": {
        "liked": {"godfather", "pulp_fiction", "forrest"},
        "disliked": {"avengers", "shining"}
    },
    "charlie": {
        "liked": {"avengers", "matrix", "shining"},
        "disliked": {"titanic", "forrest"}
    }
}

```

```

print("\n🎬 СИСТЕМА РЕКОМЕНДАЦІЙ ФІЛЬМІВ")
print("=" * 70)

```

```

def get_user_profile(user_id):
    """Створює профіль користувача на основі його вподобань"""
    if user_id not in users:
        return None

    user = users[user_id]
    profile = {
        "liked_genres": set(),
        "liked_decades": set(),
        "disliked_genres": set()
    }

```

Аналізуємо вподобані фільми

```

for movie_id in user["liked"]:
    if movie_id in movies:
        movie = movies[movie_id]
        profile["liked_genres"].update(movie["genre"])
        decade = (movie["year"] // 10) * 10
        profile["liked_decades"].add(decade)

# Аналізуємо неподобані фільми
for movie_id in user["disliked"]:
    if movie_id in movies:
        profile["disliked_genres"].update(movies[movie_id]["genre"])

return profile

def recommend_movies(user_id, count=5):
    """Рекомендує фільми користувачу"""
    if user_id not in users:
        return []

    profile = get_user_profile(user_id)
    user = users[user_id]

    # Фільми, які користувач уже бачив
    seen = user["liked"] | user["disliked"]

    # Кандидати для рекомендації
    candidates = []

    for movie_id, movie in movies.items():
        if movie_id in seen:
            continue

        # Обчислюємо оцінку для фільму
        score = 0

        # Подобаються жанри
        common_genres = len(profile["liked_genres"] & movie["genre"])

```

```

score += common_genres * 3

# Неподобаються жанри (штраф)
bad_genres = len(profile["disliked_genres"] & movie["genre"])
score -= bad_genres * 5

# Фільми з улюбленого десятиліття
movie_decade = (movie["year"] // 10) * 10
if movie_decade in profile["liked_decades"]:
    score += 2

candidates.append((movie_id, movie, score))

# Сортуюмо за оцінкою
candidates.sort(key=lambda x: x[2], reverse=True)

return candidates[:count]

def find_similar_users(user_id):
    """Знаходить користувачів зі схожими вподобаннями"""
    if user_id not in users:
        return []

    user = users[user_id]
    similarities = []

    for other_id, other in users.items():
        if other_id == user_id:
            continue

        # Обчислюємо схожість на основі спільних фільмів
        common_liked = len(user["liked"] & other["liked"])
        common_disliked = len(user["disliked"] & other["disliked"])
        common_total = common_liked + common_disliked

        # Жаккардова відстань
        total_unique = len(user["liked"] | user["disliked"] |

```

```

        other["liked"] | other["disliked"])

    if total_unique > 0:
        similarity = common_total / total_unique
    else:
        similarity = 0

    similarities.append((other_id, similarity, common_liked,
common_disliked))

    return sorted(similarities, key=lambda x: x[1], reverse=True)

# Демонстрація
print("\n👤 Профіль Аліси:")
profile = get_user_profile("alice")
print(f"  Улюблені жанри: {', '.join(profile['liked_genres'])}")
print(f"  Улюблені десятиліття: {profile['liked_decades']}")

print("\n🎬 Рекомендації для Аліси:")
recommendations = recommend_movies("alice", 3)
for movie_id, movie, score in recommendations:
    print(f"  {movie['title']} ({movie['year']}) – {', '.join(movie['genre'])}
(оцінка: {score})")

print("\n👥 Користувачі, схожі на Алісу:")
similar = find_similar_users("alice")
for other_id, similarity, common_liked, common_disliked in similar:
    print(f"  {other_id}: схожість {similarity:.2f} "
          f"(спільних лайків: {common_liked}, спільних дизлайків:
{common_disliked})")

```

Висновки: як вибрати правильну структуру даних

Задача	Рекомендована структура	Чому
Потрібен порядок елементів	Список (list)	Зберігає порядок

Задача	Рекомендована структура	Чому
		додавання
Потрібен швидкий пошук за ключем	Словник (dict)	O(1) доступ за ключем
Потрібна унікальність елементів	Множина (set)	Автоматично видаляє дублікати
Потрібні математичні операції над множинами	Множина (set)	Підтримує об'єднання, перетин, різницю
Дані не повинні змінюватися	Кортеж (tuple)	Гарантує незмінність
Потрібно рахувати частоту елементів	Counter (collections)	Спеціалізований лічильник
Потрібен словник зі значенням за замовчуванням	defaultdict (collections)	Автоматично створює відсутні ключі

У наступній лекції ми перейдемо до функцій – одного з найважливіших інструментів структурування коду та повторного використання логіки.

Питання для обговорення

1. Поясніть структуру словника як колекції "ключ-значення". Чому ключі мають бути унікальними та незмінними? Які типи даних можуть бути ключами, а які – ні?

2. У чому різниця між доступом до елемента словника через квадратні дужки (dict[key]) та метод get(key)? Чому метод get() вважається безпечнішим?

3. Проаналізуйте код:

```
d = {}
d['key'] = d.get('key', 0) + 1
print(d)
```

Що відбувається на другому рядку? Як це працює і для чого це може бути корисним?

4. Які методи словника використовуються для отримання всіх ключів, всіх значень та всіх пар ключ-значення? Наведіть приклад ітерації по словнику з використанням методу `items()`.

5. Чому множини (`set`) не зберігають порядок елементів? Яка основна перевага множин перед списками при перевірці наявності елемента?

6. Поясніть математичні операції над множинами: об'єднання (`()`), перетин (`&`), різниця (`-`), симетрична різниця (`^`). Наведіть приклади з реального життя для кожної операції.

7. У чому різниця між методами `remove()` та `discard()` для множин? Чому існують два схожі методи?

8. Напишіть програму, яка приймає два списки слів і виводить:

- Слова, які є в обох списках
- Слова, які є тільки в першому списку
- Слова, які є тільки в другому списку
- Використовуйте множини та операції над ними.

9. Що таке `defaultdict` з модуля `collections`? Чим він відрізняється від звичайного словника? Наведіть приклад, де `defaultdict` значно спрощує код.

10. Проаналізуйте код:

```
words = ['яблуко', 'банан', 'яблуко', 'апельсин', 'банан', 'банан']
counter = {}
for word in words:
    if word in counter:
        counter[word] += 1
    else:
        counter[word] = 1
```

Перепишіть цей код з використанням `defaultdict(int)`. Як зміниться код і чому він стане елегантнішим?

ЛЕКЦІЯ 9

Основи функцій у Python

План лекції

1. Поняття функції та її призначення: навіщо нам функції?

Пояснимо саму концепцію функції як підпрограми, яку можна викликати багаторазово. Розберемо принцип DRY (Don't Repeat Yourself) та переваги використання функцій: модульність, повторне використання, абстракція, тестування.

2. Синтаксис функцій: визначення, виклик, параметри та аргументи.

Детально розберемо, як створювати функції за допомогою ключового слова `def`. Розглянемо різницю між параметрами (те, що в оголошенні) та аргументами (те, що передаємо при виклику). Покажемо, як функції можуть приймати дані та як повертати результати за допомогою `return`.

3. Параметри функцій: позиційні, іменовані та значення за замовчуванням.

Розширимо розуміння параметрів. Пояснимо, як передавати аргументи за позицією та за іменем, як встановлювати значення за замовчуванням, і чому порядок параметрів важливий.

4. Область видимості змінних: локальні та глобальні.

Розберемо важливе поняття області видимості (`scope`). Пояснимо, чому змінні всередині функції не видно ззовні, і як це захищає дані. Розглянемо ключові слова `global` та `nonlocal`, але з застереженнями щодо їх використання.

1. Поняття функції та її призначення: навіщо нам функції?

Уявіть, що вам потрібно написати програму, яка обчислює площу круга. Формула проста: πr^2 . Ви пишете код:

```
radius = 5
area = 3.14159 * radius ** 2
print(f'Площа круга: {area}')
```

Але раптом вам знадобилося обчислити площу трьох різних кругів. Ви можете просто скопіювати код тричі:

```
# Круг 1
radius1 = 5
area1 = 3.14159 * radius1 ** 2
```



```
print(f'Площа круга 1: {area1}')
```

```
# Круг 2
```

```
radius2 = 7
```

```
area2 = 3.14159 * radius2 ** 2
```

```
print(f'Площа круга 2: {area2}')
```

```
# Круг 3
```

```
radius3 = 10
```

```
area3 = 3.14159 * radius3 ** 2
```

```
print(f'Площа круга 3: {area3}')
```

Це працює, але має серйозні недоліки:

- Ми повторюємо той самий код знову і знову (порушуємо принцип DRY)
- Якщо в формулі потрібно щось змінити (наприклад, точність π),

доведеться змінювати в трьох місцях

- Код стає довгим і важким для читання

Саме для таких ситуацій існують **функції**.

Що таке функція?

Функція – це іменованний блок коду, який можна викликати багаторазово.

Вона може приймати входні дані (параметри) і повертати результат. Функція – це як маленька програма всередині вашої програми.

Ось як виглядає та сама задача з функцією:

```
def circle_area(radius):
```

```
    """Обчислює площу круга за заданим радіусом"""
```

```
    pi = 3.14159
```

```
    area = pi * radius ** 2
```

```
    return area
```

```
# Використовуємо функцію багаторазово
```

```
print(f'Площа круга 1: {circle_area(5)}')
```

```
print(f'Площа круга 2: {circle_area(7)}')
```

```
print(f'Площа круга 3: {circle_area(10)}')
```

Коротко, зрозуміло, і якщо потрібно змінити формулу – змінюємо в одному місці!

Чому функції – це важливо?

1. **Повторне використання коду (Reusability):** Написали один раз – використовуємо багаторазово. Це економить час і зусилля.

2. **Модульність (Modularity):** Функції дозволяють розбити складну програму на невеликі, незалежні частини. Кожна функція робить щось одне, але робить це добре. Це називається принципом "єдиної відповідальності".

3. **Абстракція (Abstraction):** Використовуючи функцію, нам не потрібно знати, як вона працює всередині. Ми просто знаємо, що вона робить. Наприклад, ви користуєтесь функцією `print()`, не знаючи, як вона влаштована.

4. **Тестування та налагодження:** Маленькі функції легше тестувати окремо. Якщо програма не працює, ми можемо перевіряти кожен функцію ізольовано.

5. **Читабельність:** Добре названа функція робить код зрозумілим. Замість коментарів ми використовуємо назви функцій.

Принцип DRY (Don't Repeat Yourself)

Це один з найважливіших принципів програмування: "Не повторюй себе". Кожна частина знання або логіки повинна мати єдине, недвозначне представлення в системі.

ПОГАНО: повторюємо логіку

```
name1 = "Олена"  
age1 = 19  
print(f"{name1} має {age1} років")
```

```
name2 = "Іван"  
age2 = 20  
print(f"{name2} має {age2} років")
```

```
name3 = "Марія"  
age3 = 18  
print(f"{name3} має {age3} років")
```

ДОБРЕ: використовуємо функцію

```
def print_person_info(name, age):  
    """Виводить інформацію про людину"""  
    print(f"{name} має {age} років")
```

```
print_person_info("Олена", 19)
```

```
print_person_info("Іван", 20)
print_person_info("Марія", 18)
```

Коли варто створювати функцію?

- Коли ви помічаєте, що повторюєте той самий код двічі або більше
- Коли частина коду робить одну логічно завершену дію
- Коли код стає занадто довгим (більше 20-30 рядків)
- Коли вам потрібна операція, яка може стати в пригоді в інших програмах

2. Синтаксис функцій: визначення, виклик, параметри та аргументи

Тепер, коли ми зрозуміли, навіщо потрібні функції, навчимося їх створювати та використовувати.

Визначення функції (function definition)

В Python функція визначається за допомогою ключового слова `def`, за яким іде назва функції, круглі дужки з параметрами (якщо вони є) і двокрапка. Тіло функції записується з відступом.

```
def greet():
    """Проста функція без параметрів"""
    print("Привіт, світе!")
    print("Я – функція!")
```

Виклик функції (function call)

Щоб виконати функцію, потрібно її **викликати**, написавши її назву з круглими дужками:

```
# Визначення функції
def greet():
    print("Привіт, світе!")

# Виклик функції
greet() # Привіт, світе!
greet() # Можна викликати багато разів
```

Параметри та аргументи

Часто функції потребують вхідних даних. Дані, які функція отримує, називаються **параметрами** (в оголошенні) або **аргументами** (при виклику).

Функція з параметрами

```
def greet_person(name):  
    """Вітає конкретну людину"""  
    print(f"Привіт, {name}!")
```

Виклик з аргументами

```
greet_person("Олена") # Привіт, Олена!  
greet_person("Іван")  # Привіт, Іван!
```

Функція з кількома параметрами

```
def introduce(name, age, city):  
    """Представляє людину"""  
    print(f"Мене звати {name}, мені {age} років, я з {city}")
```

```
introduce("Олена", 19, "Київ")
```

```
introduce("Іван", 20, "Львів")
```

Повернення значення: оператор return

Функції можуть не тільки виконувати дії, але й **повертати** результат за допомогою оператора return. Це дозволяє використовувати результат роботи функції в подальших обчисленнях.

```
def add(a, b):  
    """Повертає суму двох чисел"""  
    result = a + b  
    return result
```

Використовуємо повернуте значення

```
sum1 = add(5, 3)  
print(sum1) # 8
```

```
sum2 = add(10, 20)
```

```
print(sum2) # 30
```

Можна використовувати результат безпосередньо

```
print(add(100, 200)) # 300
```

Функції без return

Якщо у функції немає return, вона повертає спеціальне значення None:

```
def say_hello(name):  
    print(f'Привіт, {name}!')
```

result = say_hello("Олена") # Привіт, Олена!
print(result) # None
print(type(result)) # <class 'NoneType'>

Різниця між print() та return

Це важливе розрізнення, яке часто плутають початківці:

```
def print_sum(a, b):  
    """Тільки друкує суму, але не повертає її"""  
    print(f'Сума: {a + b}')
```

```
def return_sum(a, b):  
    """Повертає суму, але нічого не друкує"""  
    return a + b
```

Використовуємо print_sum
print_sum(5, 3) # Сума: 8

Використовуємо return_sum
result = return_sum(5, 3)
print(f'Результат обчислення: {result}') # Результат обчислення: 8

Тепер ми можемо використовувати результат далі
double = return_sum(5, 3) * 2
print(double) # 16

З print_sum так не вийде
double = print_sum(5, 3) * 2 # Помилка! None * 2

Декілька значень у return

Python дозволяє повертати кілька значень. Насправді вони повертаються як кортеж:

```
def get_min_max(numbers):  
    """Повертає мінімальне та максимальне значення зі списку"""  
    return min(numbers), max(numbers)  
  
result = get_min_max([5, 2, 8, 1, 9, 3])  
print(result) # (1, 9)  
print(type(result)) # <class 'tuple'>  
  
# Можна одразу розпакувати  
min_val, max_val = get_min_max([5, 2, 8, 1, 9, 3])  
print(f"Мінімум: {min_val}, Максимум: {max_val}")
```

Docstrings – документація функцій

Хорошим тоном є документувати свої функції за допомогою рядків документації (docstrings). Вони записуються одразу після визначення функції в потрібних лапках:

```
def calculate_bmi(weight, height):  
    """  
    Обчислює індекс маси тіла (BMI).  
  
    Параметри:  
    weight (float): вага в кілограмах  
    height (float): зріст в метрах  
  
    Повертає:  
    float: індекс маси тіла  
    """  
    return weight / (height ** 2)  
  
# Документацію можна подивитися  
help(calculate_bmi)  
print(calculate_bmi.__doc__)
```

Практичний приклад: набір корисних функцій

Створимо бібліотеку корисних функцій

```
def celsius_to_fahrenheit(celsius):  
    """  
    Конвертує градуси Цельсія в Фаренгейти.  
    Формула: °F = (°C × 9/5) + 32  
    """  
    return (celsius * 9/5) + 32
```

```
def fahrenheit_to_celsius(fahrenheit):  
    """  
    Конвертує градуси Фаренгейта в Цельсії.  
    Формула: °C = (°F - 32) × 5/9  
    """  
    return (fahrenheit - 32) * 5/9
```

```
def is_even(number):  
    """Перевіряє, чи є число парним"""  
    return number % 2 == 0
```

```
def is_prime(number):  
    """Перевіряє, чи є число простим"""  
    if number < 2:  
        return False  
    for i in range(2, int(number ** 0.5) + 1):  
        if number % i == 0:  
            return False  
    return True
```

```
def factorial(n):  
    """Обчислює факторіал числа n (n!)"""  
    if n < 0:  
        return None  
    result = 1  
    for i in range(1, n + 1):  
        result *= i
```

```

    return result

# Демонстрація роботи
print("КОРИСНІ ФУНКЦІЇ")
print("=" * 50)

print(f"25°C = {celsius_to_fahrenheit(25):.1f}°F")
print(f"77°F = {fahrenheit_to_celsius(77):.1f}°C")

numbers = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
print(f"\nПарні числа: {[n for n in numbers if is_even(n)]}")
print(f"Прості числа: {[n for n in numbers if is_prime(n)]}")

print(f"\nФакторіали:")
for i in range(1, 11):
    print(f"{i}! = {factorial(i)}")

```

Практичний приклад: програма для роботи з геометричними фігурами

```

import math

def circle_area(radius):
    """Обчислює площу круга"""
    return math.pi * radius ** 2

def circle_circumference(radius):
    """Обчислює довжину кола"""
    return 2 * math.pi * radius

def rectangle_area(length, width):
    """Обчислює площу прямокутника"""
    return length * width

def rectangle_perimeter(length, width):
    """Обчислює периметр прямокутника"""
    return 2 * (length + width)

```



```

def triangle_area(base, height):
    """Обчислює площу трикутника за основою та висотою"""
    return 0.5 * base * height

def triangle_perimeter(a, b, c):
    """Обчислює периметр трикутника за трьома сторонами"""
    return a + b + c

def right_triangle_hypotenuse(a, b):
    """Обчислює гіпотенузу прямокутного трикутника"""
    return math.sqrt(a**2 + b**2)

def cylinder_volume(radius, height):
    """Обчислює об'єм циліндра"""
    return math.pi * radius**2 * height

def sphere_volume(radius):
    """Обчислює об'єм сфери"""
    return 4/3 * math.pi * radius**3

# Інтерактивна програма
def geometry_calculator():
    """Головна функція-меню для роботи з геометричними обчисленнями"""

    print("◆ ГЕОМЕТРИЧНИЙ КАЛЬКУЛЯТОР")
    print("=" * 50)

    while True:
        print("\nОберіть фігуру:")
        print("1. Круг")
        print("2. Прямокутник")
        print("3. Трикутник")
        print("4. Циліндр")
        print("5. Сфера")
        print("6. Вийти")

        choice = input("Ваш вибір (1-6): ").strip()

```

```

if choice == "1":
    radius = float(input("Радіус: "))
    print(f"\nПлоща круга: {circle_area(radius):.2f}")
    print(f"Довжина кола: {circle_circumference(radius):.2f}")

elif choice == "2":
    length = float(input("Довжина: "))
    width = float(input("Ширина: "))
    print(f"\nПлоща прямокутника: {rectangle_area(length, width):.2f}")
    print(f"Периметр прямокутника: {rectangle_perimeter(length,
width):.2f}")

elif choice == "3":
    print("\nТип трикутника:")
    print("1. За основою та висотою")
    print("2. За трьома сторонами")
    tri_choice = input("Вибір: ")

    if tri_choice == "1":
        base = float(input("Основа: "))
        height = float(input("Висота: "))
        print(f"\nПлоща трикутника: {triangle_area(base, height):.2f}")
    elif tri_choice == "2":
        a = float(input("Сторона a: "))
        b = float(input("Сторона b: "))
        c = float(input("Сторона c: "))
        print(f"\nПериметр трикутника: {triangle_perimeter(a, b, c):.2f}")

    # Перевірка, чи є трикутник прямокутним
    sides = sorted([a, b, c])
    if abs(sides[2]**2 - (sides[0]**2 + sides[1]**2)) < 0.0001:
        print("Це прямокутний трикутник!")

elif choice == "4":
    radius = float(input("Радіус основи: "))
    height = float(input("Висота: "))

```

```

print(f"\nОб'єм циліндра: {cylinder_volume(radius, height):.2f}")
print(f"Площа основи: {circle_area(radius):.2f}")

elif choice == "5":
    radius = float(input("Радіус: "))
    print(f"\nОб'єм сфери: {sphere_volume(radius):.2f}")
    print(f"Площа поверхні: {4 * circle_area(radius):.2f}")

elif choice == "6":
    print("До побачення!")
    break

else:
    print("✗ Невірний вибір")

print("-" * 30)

```

Запускаємо програму

geometry_calculator()

Цей приклад демонструє:

- Створення багатьох функцій, кожна з яких робить одну конкретну справу
- Використання return для повернення результатів
- Документування функцій (docstrings)
- Комбінування функцій у більшій програмі
- Повторне використання функцій (наприклад, circle_area()) використовується в кількох місцях)

У наступних пунктах ми розглянемо більш детально параметри функцій та області видимості змінних.

3. Параметри функцій: позиційні, іменовані та значення за замовчуванням

Коли ми створюємо функції, ми часто хочемо зробити їх гнучкими, щоб вони могли працювати з різними наборами даних. Саме для цього існують

параметри. У Python є кілька способів передавати аргументи у функції, і розуміння цих способів є ключем до написання чистого та зрозумілого коду.

Позиційні аргументи (positional arguments)

Це найпростіший і найпоширеніший спосіб передачі аргументів. Аргументи передаються в тому самому порядку, в якому визначені параметри функції.

```
def describe_person(name, age, city):  
    """Описує людину за ім'ям, віком та містом"""  
    print(f"{name}, {age} років, проживає в місті {city}")
```

Аргументи передаються в правильному порядку

```
describe_person("Олена", 19, "Київ")
```

```
describe_person("Іван", 20, "Львів")
```

Якщо переплутати порядок – отримаємо нісенітницю

```
describe_person(19, "Олена", "Київ") # 19, Олена років, проживає в місті
```

Київ? Неправильно!

Іменовані аргументи (keyword arguments)

При виклику функції ми можемо явно вказувати, якому параметру яке значення призначається, використовуючи синтаксис параметр=значення. Це робить код більш читабельним і дозволяє передавати аргументи в довільному порядку.

```
def describe_person(name, age, city):  
    print(f"{name}, {age} років, проживає в місті {city}")
```

Використання іменованих аргументів

```
describe_person(name="Олена", age=19, city="Київ")
```

```
describe_person(city="Львів", name="Іван", age=20) # порядок не важливий!
```

Можна комбінувати позиційні та іменовані аргументи

(але позиційні мають йти першими!)

```
describe_person("Марія", city="Харків", age=18) # правильно
```

describe_person(name="Марія", 18, "Харків") # ПОМИЛКА! позиційні після іменованих

Параметри зі значенням за замовчуванням

Дуже часто деякі параметри функції мають типові значення, які використовуються в більшості випадків. Ми можемо задати значення за замовчуванням прямо в оголошенні функції. Тоді при виклику ці параметри можна опустити.

```
def create_user(name, age, city="Невідомо", is_student=False):  
    """Створює профіль користувача з опціональними полями"""  
    print(f"Ім'я: {name}")  
    print(f"Вік: {age}")  
    print(f"Місто: {city}")  
    print(f"Студент: {'Так' if is_student else 'Ні'}")  
    print("-" * 20)
```

Використовуємо значення за замовчуванням

```
create_user("Олена", 19)
```

```
create_user("Іван", 20, "Львів")
```

```
create_user("Марія", 18, is_student=True)
```

```
create_user("Петро", 22, "Одеса", True)  # можна і так, але менш  
читабельно
```

Важливе застереження: значення за замовчуванням обчислюються один раз!

Це дуже важливий момент, який часто призводить до помилок. Значення за замовчуванням створюються в момент визначення функції, а не при кожному виклику. Для незмінних типів (числа, рядки, None) це не проблема, але для змінних (списки, словники) це може призвести до неочікуваної поведінки:

НЕПРАВИЛЬНО! Ніколи не використовуйте змінні типи як значення за замовчуванням

```
def add_task(task, task_list=[]):  
    """Додає завдання до списку (з підводним каменем!)"""  
    task_list.append(task)  
    return task_list
```

```
print(add_task("купити молоко"))  # ['купити молоко']
```

```
print(add_task("зробити домашку"))  # ['купити молоко', 'зробити  
домашку'] – список не скинувся!
```

Чому? Тому що список task_list створюється один раз і використовується при всіх викликах

ПРАВИЛЬНО

```
def add_task_correct(task, task_list=None):  
    """Додає завдання до списку (правильна версія)"""  
    if task_list is None:  
        task_list = []  
    task_list.append(task)  
    return task_list
```

```
print(add_task_correct("купити молоко")) # ['купити молоко']
```

```
print(add_task_correct("зробити домашку")) # ['зробити домашку'] –  
новий список!
```

Правила використання параметрів

В Python діє чіткий порядок при оголошенні функції:

1. Спочатку звичайні позиційні параметри
2. Потім параметри зі значенням за замовчуванням
3. Потім спеціальні параметри (*args, **kwargs – про це в наступній лекції)

Правильний порядок

```
def correct_function(a, b, c=10, d=20):  
    pass
```

Неправильний порядок (помилка)

```
# def wrong_function(a=10, b, c): # SyntaxError: non-default argument follows  
# default argument  
#     pass
```

Практичний приклад: гнучка функція для створення листів

```
def generate_email(to, subject, body, cc=None, bcc=None, priority="normal",  
attachments=None):
```

```
    """
```

Генерує email з різними опціональними параметрами.

Параметри:

to – обов’язковий, одержувач

subject – обов’язковий, тема

body – обов’язковий, тіло листа

cc – опціональний, копія

bcc – опціональний, прихована копія

priority – опціональний, пріоритет (normal, high, low)

attachments – опціональний, список вкладень

"""

```
print("=" * 60)
```

```
print(f"ДО: {to}")
```

```
if cc:
```

```
    print(f"КОПІЯ: {cc}")
```

```
if bcc:
```

```
    print(f"ПРИХОВАНА КОПІЯ: {bcc}")
```

```
print(f"ТЕМА: {subject}")
```

```
print(f"ПРІОРИТЕТ: {priority.upper()}")
```

```
print("-" * 60)
```

```
print(body)
```

```
if attachments:
```

```
    print("-" * 60)
```

```
    print(f"ВКЛАДЕННЯ: {', '.join(attachments)}")
```

```
print("=" * 60)
```

Використання з різними наборами параметрів

```
generate_email(
```

```
    to="student@example.com",
```

```
    subject="Запрошення на лекцію",
```

```
    body="Шановний студенте, запрошуємо вас на відкриту лекцію з  
Python."
```

```
)
```

```
print("\n")
```

```
generate_email(
```

```
    to="teacher@example.com",
```

```
    cc="dean@example.com",
```

```

subject="Звіт про успішність",
body="Надсилаю звіт про успішність студентів за семестр.",
priority="high",
attachments=["grades.csv", "summary.pdf"]
)

```

4. Область видимості змінних: локальні та глобальні

Коли ми працюємо з функціями, дуже важливо розуміти, де які змінні існують і чи можна до них звернутися. Це поняття називається **областю видимості (scope)**.

Локальні змінні

Змінні, створені всередині функції, називаються **локальними**. Вони існують тільки під час виконання функції і недоступні ззовні.

```

def my_function():
    x = 10 # локальна змінна
    print(f'Всередині функції: x = {x}')

```

```

my_function() # Всередині функції: x = 10
# print(x) # NameError: name 'x' is not defined – змінна не існує поза функцією

```

Кожен виклик функції створює нове локальне середовище. Змінні з різних викликів не заважають одна одній:

```

def increment(number):
    result = number + 1
    return result

a = increment(5)
b = increment(10)
print(a, b) # 6 11 – кожен виклик мав свою локальну змінну result

```

Глобальні змінні

Змінні, визначені поза будь-якою функцією, називаються **глобальними**. Вони доступні з будь-якого місця програми.


```
global_var = 100 # глобальна змінна
```

```
def show_global():
```

```
    print(f'Всередині функції: {global_var}') # можемо читати глобальну змінну
```

```
show_global() # Всередині функції: 100
```

```
print(f'Ззовні: {global_var}') # Ззовні: 100
```

Але є нюанс! Якщо ми спробуємо **змінити** глобальну змінну всередині функції, Python створить нову локальну змінну з тим самим іменем, залишивши глобальну без змін:

```
global_var = 100
```

```
def try_to_change():
```

```
    global_var = 200 # це НЕ зміна глобальної, а створення НОВОЇ локальної!
```

```
    print(f'Всередині: {global_var}')
```

```
try_to_change() # Всередині: 200
```

```
print(f'Ззовні: {global_var}') # Ззовні: 100 – глобальна не змінилася!
```

Ключове слово **global**

Якщо нам дійсно потрібно змінити глобальну змінну всередині функції, ми маємо явно оголосити це за допомогою ключового слова **global**:

```
counter = 0
```

```
def increment_counter():
```

```
    global counter # кажемо Python, що ми хочемо використовувати глобальну змінну
```

```
    counter += 1
```

```
    print(f'Лічильник збільшено до {counter}')
```

```
increment_counter() # Лічильник збільшено до 1
```

```
increment_counter() # Лічильник збільшено до 2
```

```
increment_counter() # Лічильник збільшено до 3
```

```
print(f'Кінцеве значення: {counter}') # Кінцеве значення: 3
```

Чому варто уникати global?

Хоча global існує, більшість досвідчених програмістів радять уникати його використання. Чому?

1. **Важко відстежувати:** Функція може змінювати змінні в будь-якому місці програми, що ускладнює розуміння коду.

2. **Важко тестувати:** Функція залежить від зовнішнього стану, який важко контролювати.

3. **Порушує принцип ізоляції:** Функції мають бути "чистими" – залежати тільки від своїх параметрів і не мати побічних ефектів.

Кращий підхід – передавати значення через параметри і повертати результати:

```
# ПОГАНО
```

```
total = 0
```

```
def add_bad(x):
```

```
    global total
```

```
    total += x
```

```
add_bad(5)
```

```
add_bad(10)
```

```
print(total) # 15
```

```
# ДОБРЕ
```

```
def add_good(total, x):
```

```
    return total + x
```

```
total = 0
```

```
total = add_good(total, 5)
```

```
total = add_good(total, 10)
```

```
print(total) # 15
```

Вкладені функції та nonlocal

У Python функції можна визначати всередині інших функцій. Тоді виникає питання: як змінювати змінні зовнішньої функції? Для цього використовується ключове слово nonlocal:

```
def outer_function():
    message = "Зовнішнє повідомлення"

    def inner_function():
        nonlocal message # кажемо, що хочемо змінити змінну з outer_function
        message = "Змінено внутрішньою функцією"
        print(f'Всередині: {message}')

    inner_function()
    print(f'Зовні: {message}')

outer_function()
# Всередині: Змінено внутрішньою функцією
# Зовні: Змінено внутрішньою функцією
```

Практичний приклад: створення лічильника з використанням замикання

```
def create_counter():
    """
    Створює лічильник, який запам'ятовує свій стан.
    Демонстрація замикання (closure) без використання global.
    """
    count = 0

    def increment():
        nonlocal count
        count += 1
        return count

    def decrement():
        nonlocal count
        count -= 1
        return count

    def get_value():
        return count
```

```

# Повертаємо функції для роботи з лічильником
return {"increment": increment, "decrement": decrement, "value": get_value}

# Створюємо два незалежних лічильники
counter1 = create_counter()
counter2 = create_counter()

print("Лічильник 1:")
print(counter1["increment"]()) # 1
print(counter1["increment"]()) # 2
print(counter1["increment"]()) # 3
print(f"Значення: {counter1['value']}") # Значення: 3

print("\nЛічильник 2:")
print(counter2["increment"]()) # 1
print(counter2["decrement"]()) # 0
print(counter2["increment"]()) # 1
print(f"Значення: {counter2['value']}") # Значення: 1

# Лічильники не заважають один одному!

```

Практичний приклад: менеджер конфігурації

```

def create_config_manager():
    """
    Створює менеджер конфігурації, який зберігає налаштування.
    Демонструє безпечне використання замикань замість глобальних
    змінних.
    """
    config = {
        "theme": "light",
        "language": "uk",
        "font_size": 12,
        "notifications": True
    }

    def get(key, default=None):
        """Отримує значення за ключем"""

```

```

    return config.get(key, default)

def set(key, value):
    """Встановлює значення за ключем"""
    config[key] = value
    return True

def update(new_config):
    """Оновлює кілька налаштувань одночасно"""
    config.update(new_config)
    return True

def reset():
    """Скидає налаштування до стандартних"""
    nonlocal config
    config = {
        "theme": "light",
        "language": "uk",
        "font_size": 12,
        "notifications": True
    }
    return True

def show_all():
    """Показує всі налаштування"""
    return config.copy()

return {"get": get, "set": set, "update": update, "reset": reset, "show_all":
show_all}

# Створюємо менеджер конфігурації
config = create_config_manager()

print("Початкова конфігурація:")
print(config["show_all"]())

print("\nЗмінюємо налаштування:")

```

```
config["set"]("theme", "dark")
config["set"]("font_size", 14)
config["update"]({"language": "en", "notifications": False})
```

```
print("Оновлена конфігурація:")
print(config["show_all"]())
```

```
print(f"\nПоточний розмір шрифту: {config['get']('font_size')}")
print(f"Невідоме налаштування: {config['get']('volume', 'не встановлено')}")
```

```
print("\nСкидаємо налаштування:")
config["reset]()
print(config["show_all"]())
```

Порівняння підходів до збереження стану

Підхід	Переваги	Недоліки
Глобальні змінні	Простота для малих програм	Важко відстежувати, небезпечно
Параметри та повернення	Чистота, передбачуваність	Може бути громіздко
Замикання (closure)	Інкапсуляція, зручність	Складніше для розуміння
Класи (будемо вивчати)	Найкраща організація	Потребує знання ООП

Типові помилки з областю видимості

1. Забування про локальність змінних:

```
def process_data(data):
    result = [] # локальна змінна
    for item in data:
        result.append(item * 2)
    return result
```

```
my_data = [1, 2, 3]
processed = process_data(my_data)
# print(result) # NameError: result не визначена поза функцією
```

2. Неочікуване створення локальної змінної:

```
x = 10
def modify_x():
    x = x + 1 # UnboundLocalError: local variable 'x' referenced before
assignment
# Python думає, що x локальна, але вона ще не визначена!
```

```
modify_x()
```

3. Використання змінних зовнішнього скоупу без необхідності:

```
# ПОГАНО
default_name = "Гість"
def greet(name=default_name): # default_name береться з глобального
скоупу
    print(f"Привіт, {name}!")

# КРАЩЕ
def greet(name="Гість"): # значення за замовчуванням в самій функції
    print(f"Привіт, {name}!")
```

Висновки

- Локальні змінні існують тільки всередині функцій і захищають дані від неочікуваних змін.
- Глобальні змінні доступні всюди, але їх використання робить код менш передбачуваним.
- `global` дозволяє змінювати глобальні змінні, але краще цього уникати.
- `nonlocal` потрібен для роботи зі змінними зовнішніх (але не глобальних) функцій.
- Замикання (closures) дозволяють створювати функції з власним станом без використання глобальних змінних.

У наступній лекції ми розглянемо більш просунуті можливості функцій: функції з довільною кількістю аргументів, лямбда-функції, функції вищого порядку та рекурсію.

Питання для обговорення

1. Поясніть принцип DRY (Don't Repeat Yourself). Як функції допомагають дотримуватися цього принципу? Наведіть приклад коду з повторенням і його ж, але з використанням функції.

2. У чому різниця між параметрами та аргументами функції? Наведіть приклад функції з двома параметрами та двох різних викликів цієї функції.

3. Що таке оператор return? Чим функція, яка повертає значення, відрізняється від функції, яка тільки виконує дії (наприклад, друкує)? Чому функції без return повертають None?

4. Проаналізуйте код:

```
def add(a, b):  
    result = a + b
```

```
sum = add(5, 3)  
print(sum)
```

Чому sum дорівнює None? Як виправити функцію, щоб вона повертала правильний результат?

5. Поясніть різницю між позиційними та іменованими аргументами. Чому іменовані аргументи роблять код більш читабельним? Наведіть приклад.

6. Що таке параметри зі значенням за замовчуванням? Чому вони корисні? Яке правило визначення таких параметрів (чому вони мають бути після звичайних параметрів)?

7. Чому наступний код працює неочікувано?

```
def add_task(task, task_list=[]):  
    task_list.append(task)  
    return task_list
```

```
print(add_task("task1"))  
print(add_task("task2"))
```

Як правильно реалізувати функцію, яка додає завдання до списку, з можливістю передати існуючий список або створити новий?

8. Поясніть поняття області видимості змінних. У чому різниця між локальними та глобальними змінними? Чому вважається поганою практикою використовувати глобальні змінні всередині функцій?

9. Проаналізуйте код:


```
x = 10
def func():
    x = 20
    print(x)
func()
print(x)
```

Чому перший print виводить 20, а другий 10? Що станеться, якщо прибрати рядок `x = 20` всередині функції?

10. Напишіть функцію, яка приймає список чисел і повертає словник з ключами `sum`, `average`, `min`, `max`. Використовуйте вбудовані функції `sum()`, `min()`, `max()`.

ЛЕКЦІЯ 10

Розширені можливості функцій

План лекції

1. ****Функції з довільною кількістю аргументів: *args та kwargs.**

Розберемо, як створювати функції, які можуть приймати будь-яку кількість позиційних аргументів за допомогою *args та будь-яку кількість іменованих аргументів за допомогою **kwargs. Це надзвичайно корисно для створення гнучких інтерфейсів та обгорткових функцій.

2. Лямбда-функції: анонімні функції на один рядок.

Познайомимося з лямбда-функціями – компактним способом створення простих функцій "на льоту". Розберемо, де вони доречні, а де краще використовувати звичайні функції. Покажемо, як лямбди працюють з функціями вищого порядку.

3. Функції вищого порядку: map, filter, reduce.

Розглянемо функції, які приймають інші функції як аргументи. Детально розберемо роботу з map(), filter() та reduce() – елегантні способи обробки колекцій без явних циклів. Покажемо, як вони поєднуються з лямбда-функціями.

4. Рекурсія: функція, що викликає саму себе.

Пояснимо концепцію рекурсії – методу розв'язання задач, де функція викликає саму себе. Розберемо класичні приклади: факторіал, числа Фібоначчі, обхід дерев. Обговоримо переваги та недоліки рекурсії, проблему переповнення стеку та хвостову рекурсію.

****1. Функції з довільною кількістю аргументів: *args та kwargs**

Іноді ми створюємо функції, де заздалегідь не знаємо, скільки аргументів буде передано. Наприклад, функція для обчислення суми чисел – ми хочемо мати можливість передати 2, 3, 10 або навіть 100 чисел. Або функція для форматування рядка, де кількість параметрів може змінюватися. У таких випадках нам допомагають спеціальні синтаксичні конструкції: *args та **kwargs.

**Позиційні аргументи довільної довжини: args*

Символ * перед іменем параметра (зазвичай використовують назву args, але можна будь-яку) дозволяє функції приймати будь-яку кількість позиційних аргументів. Всі передані аргументи збираються в кортеж.

```
def print_all(*args):  
    """Виводить всі отримані аргументи"""  
    print(f'Отримано {len(args)} аргументів: {args}')
```

```

for i, arg in enumerate(args, 1):
    print(f'Аргумент {i}: {arg}')

print_all(1, 2, 3)
print_all("яблуко", "банан", "апельсин", "груша")
print_all() # можна і без аргументів

```

Корисний приклад: функція для обчислення суми

```

def sum_all(*args):
    """Обчислює суму всіх переданих чисел"""
    total = 0
    for num in args:
        total += num
    return total

print(sum_all(1, 2, 3))          # 6
print(sum_all(10, 20, 30, 40))  # 100
print(sum_all(1.5, 2.5, 3.5))   # 7.5

# Можна комбінувати зі звичайними параметрами
def greet_with_items(greeting, *items):
    """Вітає з переліком предметів"""
    print(f'{greeting}! Сьогодні у списку:')
    for item in items:
        print(f' - {item}')

```

```
greet_with_items("Привіт", "хліб", "молоко", "яйця")
```

****Іменовані аргументи довільної довжини: kwargs**

Два символи ****** перед іменем параметра (зазвичай kwargs) дозволяють функції приймати будь-яку кількість іменованих аргументів. Всі передані аргументи збираються в словник.

```

def print_user_info(**kwargs):
    """Виводить інформацію про користувача з довільними полями"""
    print("Інформація про користувача:")
    for key, value in kwargs.items():
        print(f' {key}: {value}')

```

```
print_user_info(name="Олена", age=19, city="Київ", occupation="студент")
print_user_info(username="ivan123", email="ivan@example.com")
```

****Комбінування *args та kwargs**

Можна використовувати обидва типи параметрів разом, але важливо дотримуватися порядку: спочатку звичайні параметри, потім *args, потім **kwargs.

```
def advanced_function(required, *args, default_param="стандарт", **kwargs):
    """Демонструє комбінування різних типів параметрів"""
    print(f'Обов'язковий параметр: {required}')
    print(f'Позиційні аргументи (*args): {args}')
    print(f'Параметр за замовчуванням: {default_param}')
    print(f'Іменовані аргументи (**kwargs): {kwargs}')
```

```
advanced_function("обов'язково", 1, 2, 3, name="Олена", age=19)
```

Розпакування аргументів при виклику функції

Те саме, що ми робимо з * та ** в оголошенні функції, можна робити і при її виклику – це називається **розпакуванням аргументів**:

```
def greet(name, age, city):
    print(f'{name}, {age} років, м. {city}')
```

Звичайний виклик

```
greet("Олена", 19, "Київ")
```

Розпакування списку/кортежу

```
person_info = ["Іван", 20, "Львів"]
```

```
greet(*person_info) # * розпаковує список в окремі аргументи
```

Розпакування словника

```
person_dict = {"name": "Марія", "age": 18, "city": "Харків"}
```

```
greet(**person_dict) # ** розпаковує словник в іменовані аргументи
```

Практичний приклад: гнучкий декоратор для логування

```
def log_function_call(func):
```

```
"""Декоратор для логування викликів функцій (демонстрація *args,
**kwargs)"""
```

```
def wrapper(*args, **kwargs):
    print(f"Виклик функції {func.__name__}")
    if args:
        print(f"Позиційні аргументи: {args}")
    if kwargs:
        print(f"Іменовані аргументи: {kwargs}")
    result = func(*args, **kwargs)
    print(f"Результат: {result}")
    return result
return wrapper
```

```
@log_function_call
```

```
def add(a, b):
    return a + b
```

```
@log_function_call
```

```
def greet(name, greeting="Привіт"):
    return f"{greeting}, {name}!"
```

```
add(5, 3)
```

```
greet("Олена", greeting="Добрий вечір")
```

Практичний приклад: функція для створення SQL-запитів

```
def create_insert_query(table_name, **fields):
```

```
    """
```

```
    Створює SQL-запит INSERT на основі переданих полів.
```

```
    **kwargs дозволяє передати будь-які поля для вставки.
```

```
    """
```

```
    columns = ', '.join(fields.keys())
```

```
    values = ', '.join([f"{'v'}" if isinstance(v, str) else str(v) for v in
fields.values()])
```

```
    query = f"INSERT INTO {table_name} ({columns}) VALUES ({values});"
```

```
    return query
```

```
def create_select_query(table_name, *columns, **conditions):
```

```
"""
```

Створює SQL-запит SELECT з умовами.

*columns – які колонки вибрати (якщо порожньо, то *)

**conditions – умови WHERE

```
"""
```

```
if not columns:
```

```
    columns_part = "*"
```

```
else:
```

```
    columns_part = ', '.join(columns)
```

```
query = f"SELECT {columns_part} FROM {table_name}"
```

```
if conditions:
```

```
    where_clauses = []
```

```
    for key, value in conditions.items():
```

```
        if isinstance(value, str):
```

```
            where_clauses.append(f"{key} = '{value}'")
```

```
        else:
```

```
            where_clauses.append(f"{key} = {value}")
```

```
    query += " WHERE " + " AND ".join(where_clauses)
```

```
    return query + ";"
```

Демонстрація

```
print("INSERT запити:")
```

```
print(create_insert_query("users", name="Олена", age=19, city="Київ"))
```

```
print(create_insert_query("products", name="Ноутбук", price=25000,  
in_stock=True))
```

```
print("\nSELECT запити:")
```

```
print(create_select_query("users", "name", "age"))
```

```
print(create_select_query("users", "name", "age", city="Київ", age=19))
```

```
print(create_select_query("products", "name", "price", price=25000))
```

2. Лямбда-функції: анонімні функції на один рядок

У Python є спосіб створювати маленькі, одноразові функції без використання `def` – це **лямбда-функції**. Вони також називаються анонімними функціями, оскільки не мають власного імені.

Синтаксис лямбда-функцій

`lambda` параметри: вираз

Лямбда-функція:

- Може приймати будь-яку кількість параметрів
- Повертає результат обчислення виразу
- Не може містити складних конструкцій (умови можна, але обмежено)
- Не може містити циклів або множинних інструкцій

Прості приклади

Звичайна функція

```
def square(x):  
    return x ** 2
```

Лямбда-функція

```
square_lambda = lambda x: x ** 2
```

```
print(square(5))      # 25  
print(square_lambda(5)) # 25
```

Лямбда з кількома параметрами

```
add = lambda a, b: a + b  
print(add(10, 20)) # 30
```

Лямбда з умовою

```
max_value = lambda a, b: a if a > b else b  
print(max_value(15, 25)) # 25
```

Де насправді використовуються лямбди?

Лямбда-функції найчастіше використовуються там, де потрібна проста функція на один раз, особливо як аргумент для інших функцій.

1. Сортування списків з ключем:

Список кортежів (ім'я, вік)

```
students = [  
    ("Олена", 19),  
    ("Іван", 20),  
    ("Марія", 18),  
    ("Петро", 21)  
]
```

Сортування за віком (другий елемент)

```
students_sorted = sorted(students, key=lambda student: student[1])  
print(students_sorted) # [('Марія', 18), ('Олена', 19), ('Іван', 20), ('Петро',  
21)]
```

Сортування за іменем

```
students_sorted = sorted(students, key=lambda student: student[0])  
print(students_sorted) # [('Іван', 20), ('Марія', 18), ('Олена', 19), ('Петро',  
21)]
```

Складніший приклад: список словників

```
people = [  
    {"name": "Олена", "age": 19, "city": "Київ"},  
    {"name": "Іван", "age": 20, "city": "Львів"},  
    {"name": "Марія", "age": 18, "city": "Харків"}  
]
```

Сортування за віком

```
people_sorted = sorted(people, key=lambda p: p["age"])  
print(people_sorted)
```

Сортування за містом (за алфавітом)

```
people_sorted = sorted(people, key=lambda p: p["city"])  
print(people_sorted)
```

2. Робота з функціями вищого порядку (про них детально в наступному пункті):

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Квадрати чисел за допомогою map


```
squares = list(map(lambda x: x ** 2, numbers))
print(squares) # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Фільтрація парних чисел

```
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers) # [2, 4, 6, 8, 10]
```

Комбінування map і filter

Квадрати тільки непарних чисел

```
odd_squares = list(map(lambda x: x ** 2, filter(lambda x: x % 2 != 0,
numbers)))
print(odd_squares) # [1, 9, 25, 49, 81]
```

Лямбди для створення простих функцій "на льоту"

Обробка рядків

```
words = ["яблуко", "банан", "апельсин", "груша"]
```

Довжина кожного слова

```
word_lengths = list(map(lambda word: len(word), words))
print(word_lengths) # [6, 5, 8, 5]
```

Слова у верхньому регістрі

```
upper_words = list(map(lambda word: word.upper(), words))
print(upper_words) # ['ЯБЛУКО', 'БАНАН', 'АПЕЛЬСИН', 'ГРУША']
```

Перевірка, чи слово починається з голосної

```
vowels = "аеіоіуяюєї"
starts_with_vowel = list(map(lambda word: word[0].lower() in vowels, words))
print(starts_with_vowel) # [True, False, True, False]
```

Коли варто використовувати лямбди, а коли – звичайні функції?

Ситуація	Краще використовувати
Проста логіка на один рядок, використовується один раз	Лямбда

Ситуація	Краще використовувати
Функція потрібна як аргумент для map/filter/sorted	Лямбда
Складна логіка, кілька рядків коду	Звичайна функція
Функція використовується багаторазово	Звичайна функція
Потрібні цикли, складні умови	Звичайна функція
Потрібна документація (docstring)	Звичайна функція

Практичний приклад: обробка даних з інтернет-магазину

Дані про товари в інтернет-магазині

```
products = [
    {"name": "Ноутбук", "price": 25000, "quantity": 5, "category":
"електроніка"},
    {"name": "Мишка", "price": 500, "quantity": 20, "category": "електроніка"},
    {"name": "Книга Python", "price": 800, "quantity": 10, "category": "книги"},
    {"name": "Навушники", "price": 1500, "quantity": 8, "category":
"електроніка"},
    {"name": "Щоденник", "price": 120, "quantity": 50, "category":
"канцтовари"},
    {"name": "Ручка", "price": 30, "quantity": 100, "category": "канцтовари"},
    {"name": "Монітор", "price": 8000, "quantity": 3, "category":
"електроніка"}
]
```

```
print("🛒 АНАЛІЗ ТОВАРІВ ІНТЕРНЕТ-МАГАЗИНУ")
print("=" * 60)
```

1. Сортування за ціною (від дешевих до дорогих)

```
cheapest_first = sorted(products, key=lambda p: p["price"])
print("\n📋 Товари за зростанням ціни:")
for p in cheapest_first[:5]:
    print(f" {p['name']}: {p['price']} грн")
```

2. Товари з ціною > 1000

```
expensive = list(filter(lambda p: p["price"] > 1000, products))
print(f"\n💰 Товарів дорожче 1000 грн: {len(expensive)}")
```

3. Загальна вартість всіх товарів на складі

```
total_value = sum(map(lambda p: p["price"] * p["quantity"], products))
print(f"💰 Загальна вартість товарів: {total_value} грн")
```

4. Середня ціна товарів у кожній категорії

```
categories = set(map(lambda p: p["category"], products))
print(f"\n📊 Середня ціна за категоріями:")
for category in categories:
    category_products = list(filter(lambda p: p["category"] == category,
products))
    avg_price = sum(map(lambda p: p["price"], category_products)) /
len(category_products)
    print(f" {category}: {avg_price:.2f} грн")
```

5. Назви товарів, яких мало на складі (quantity < 5)

```
low_stock = list(filter(lambda p: p["quantity"] < 5, products))
low_stock_names = list(map(lambda p: p["name"], low_stock))
print(f"\n⚠️ Товари з малим залишком: {', '.join(low_stock_names)}")
```

6. Знижка 10% на всі товари з категорії "електроніка"

```
electronics_with_discount = list(map(
    lambda p: {**p, "price": round(p["price"] * 0.9, 2)} if p["category"] ==
"електроніка" else p,
products
))
print(f"\n📦 Ціни на електроніку зі знижкою 10%:")
for p in filter(lambda p: p["category"] == "електроніка",
electronics_with_discount):
    original = next(prod for prod in products if prod["name"] == p["name"])
    print(f" {p['name']}: {original['price']} грн → {p['price']} грн")
```

Обмеження лямбда-функцій

Лямбди не можуть містити складних конструкцій:

Це ПРАЦЮЄ (простий вираз)

```
simple = lambda x: x ** 2
```

Це теж ПРАЦЮЄ (тернарний оператор)

```
with_condition = lambda x: "парне" if x % 2 == 0 else "непарне"
```

Це НЕ ПРАЦЮЄ (неможливо використати print)

```
# bad_lambda = lambda x: print(x); x ** 2 # SyntaxError
```

Це НЕ ПРАЦЮЄ (не можна мати кілька виразів)

```
# bad_lambda = lambda x: x += 1; return x # SyntaxError
```

Якщо потрібна складна логіка – використовуйте звичайну функцію

```
def complex_function(x):  
    print(f"Обробляємо {x}")  
    x += 1  
    return x ** 2
```

Практичний приклад: створення простих калькуляторів

Створюємо словник математичних операцій за допомогою лямбд

```
operations = {  
    "add": lambda a, b: a + b,  
    "subtract": lambda a, b: a - b,  
    "multiply": lambda a, b: a * b,  
    "divide": lambda a, b: a / b if b != 0 else "Помилка: ділення на нуль",  
    "power": lambda a, b: a ** b,  
    "mod": lambda a, b: a % b,  
    "min": lambda a, b: a if a < b else b,  
    "max": lambda a, b: a if a > b else b  
}
```

Використовуємо калькулятор

```
print("□ КАЛЬКУЛЯТОР")  
print("=" * 40)
```

```

x, y = 10, 3
print(f"Числа: {x} та {y}\n")

for op_name, op_func in operations.items():
    result = op_func(x, y)
    print(f"{op_name:10}: {result}")

# Інтерактивна версія
print("\n" + "=" * 40)
print("Інтерактивний режим:")

while True:
    expr = input("\nВведіть операцію (наприклад, add 5 3) або 'exit': ").strip()
    if expr == "exit":
        break

    parts = expr.split()
    if len(parts) != 3:
        print("Формат: операція число1 число2")
        continue

    op, a_str, b_str = parts
    if op not in operations:
        print(f"Невідома операція. Доступні: {', '.join(operations.keys())}")
        continue

    try:
        a, b = float(a_str), float(b_str)
        result = operations[op](a, b)
        print(f"Результат: {result}")
    except ValueError:
        print("Помилка: введіть числа")

```

У наступному пункті ми детально розглянемо функції вищого порядку – `map`, `filter`, `reduce` – які дозволяють обробляти колекції елегантним і функціональним способом.

3. Функції вищого порядку: map, filter, reduce

У Python функції є об'єктами першого класу. Це означає, що їх можна передавати як аргументи іншим функціям, повертати з функцій, зберігати в змінних. Функції, які приймають інші функції як аргументи або повертають функції, називаються **функціями вищого порядку**. Три класичні приклади таких функцій – map(), filter() та reduce().

Функція map() – перетворення кожного елемента

Функція map() застосовує задану функцію до кожного елемента ітерованого об'єкта (списку, кортежу тощо) і повертає ітератор з результатами.

Синтаксис: map(функція, ітерований_об'єкт)

Приклад: піднесення кожного числа до квадрату

```
numbers = [1, 2, 3, 4, 5]
```

Без map (цикл)

```
squares_loop = []
```

```
for n in numbers:
```

```
    squares_loop.append(n ** 2)
```

```
print(squares_loop) # [1, 4, 9, 16, 25]
```

З map та звичайною функцією

```
def square(x):
```

```
    return x ** 2
```

```
squares_map = list(map(square, numbers))
```

```
print(squares_map) # [1, 4, 9, 16, 25]
```

З map та лямбда-функцією (найчастіше)

```
squares_lambda = list(map(lambda x: x ** 2, numbers))
```

```
print(squares_lambda) # [1, 4, 9, 16, 25]
```

map() з кількома ітерованими об'єктами

map() може приймати кілька ітерованих об'єктів. Тоді функція повинна приймати стільки параметрів, скільки об'єктів передано.

Додавання відповідних елементів двох списків

```
a = [1, 2, 3, 4, 5]
```

```
b = [10, 20, 30, 40, 50]
```

```
sums = list(map(lambda x, y: x + y, a, b))
```

```
print(sums) # [11, 22, 33, 44, 55]
```

Перемноження елементів трьох списків

```
c = [100, 200, 300]
```

```
products = list(map(lambda x, y, z: x * y * z, a[:3], b[:3], c))
```

```
print(products) # [1000, 8000, 27000] (1*10*100, 2*20*200, 3*30*300)
```

Функція filter() – фільтрація елементів

Функція filter() створює ітератор з елементів, для яких задана функція повернула True.

Синтаксис: filter(функція, ітерований_об'єкт)

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Без filter (цикл)

```
even_loop = []
```

```
for n in numbers:
```

```
    if n % 2 == 0:
```

```
        even_loop.append(n)
```

```
print(even_loop) # [2, 4, 6, 8, 10]
```

З filter та звичайною функцією

```
def is_even(x):
```

```
    return x % 2 == 0
```

```
even_filter = list(filter(is_even, numbers))
```

```
print(even_filter) # [2, 4, 6, 8, 10]
```

З filter та лямбда-функцією

```
even_lambda = list(filter(lambda x: x % 2 == 0, numbers))
```

```
print(even_lambda) # [2, 4, 6, 8, 10]
```

Фільтрація рядків за довжиною

```
words = ["яблуко", "банан", "ананас", "груша", "апельсин", "кит"]
long_words = list(filter(lambda word: len(word) >= 6, words))
print(long_words) # ['яблуко', 'банан', 'ананас', 'апельсин']
```

Комбінування map() та filter()

Ці функції чудово поєднуються для створення ланцюжків обробки даних:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Квадрати тільки парних чисел

Спосіб 1: filter потім map

```
even_squares = list(map(lambda x: x ** 2, filter(lambda x: x % 2 == 0,
numbers)))
print(even_squares) # [4, 16, 36, 64, 100]
```

Спосіб 2: зрозуміліший через list comprehension (альтернатива)

```
even_squares_comp = [x ** 2 for x in numbers if x % 2 == 0]
print(even_squares_comp) # [4, 16, 36, 64, 100]
```

Складніший приклад: з рядками

```
words = ["python", "java", "javascript", "c++", "ruby", "php"]
```

Довжини слів, які починаються з голосної

```
vowels = "aeiou"
```

```
result = list(map(
    lambda word: len(word),
    filter(lambda word: word[0] in vowels, words)
))
```

print(result) # [6, 10] (для "python"? ні, python починається з приголосної, javascript – з приголосної? ні, javascript теж з приголосної)

Виправимо:

```
words_with_vowels = ["apple", "orange", "python", "java", "javascript"]
result = list(map(
    lambda word: len(word),
    filter(lambda word: word[0] in "aeiou", words_with_vowels)
))
print(result) # [5, 6] (apple, orange)
```


Функція `reduce()` – зведення до одного значення

Функція `reduce()` знаходиться в модулі `functools`. Вона послідовно застосовує функцію до елементів, зводячи весь список до одного значення.

```
from functools import reduce
```

```
# Синтаксис: reduce(функція, ітерований_об'єкт, [початкове_значення])
```

```
# Сума чисел
```

```
numbers = [1, 2, 3, 4, 5]
```

```
# Без reduce (цикл)
```

```
total_loop = 0
```

```
for n in numbers:
```

```
    total_loop += n
```

```
print(total_loop) # 15
```

```
# 3 reduce
```

```
total_reduce = reduce(lambda acc, x: acc + x, numbers)
```

```
print(total_reduce) # 15
```

```
# Як це працює:
```

```
# 1: acc=1, x=2 → 3
```

```
# 2: acc=3, x=3 → 6
```

```
# 3: acc=6, x=4 → 10
```

```
# 4: acc=10, x=5 → 15
```

```
# 3 початковим значенням
```

```
total_with_start = reduce(lambda acc, x: acc + x, numbers, 10)
```

```
print(total_with_start) # 25 (10 + 15)
```

Інші приклади використання `reduce()`

```
from functools import reduce
```

```
# Знаходження максимального елемента
```

```
numbers = [42, 17, 89, 33, 56, 91, 24]
```

```
max_value = reduce(lambda acc, x: acc if acc > x else x, numbers)
```

```
print(f"Максимум: {max_value}") # 91
```

```
# Добуток всіх чисел
```

```
product = reduce(lambda acc, x: acc * x, range(1, 6)) # 1*2*3*4*5
```

```
print(f"Добуток: {product}") # 120
```

```
# Конкатенація рядків
```

```
words = ["Python", " ", "це", " ", "чудово"]
```

```
sentence = reduce(lambda acc, w: acc + w, words)
```

```
print(sentence) # Python це чудово
```

```
# Створення словника зі списку пар
```

```
pairs = [("a", 1), ("b", 2), ("c", 3)]
```

```
dictionary = reduce(lambda acc, pair: {**acc, pair[0]: pair[1]}, pairs, {})
```

```
print(dictionary) # {'a': 1, 'b': 2, 'c': 3}
```

Порівняння підходів

```
from functools import reduce
```

```
# Задача: знайти суму квадратів парних чисел
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
# 1. Традиційний цикл
```

```
result_loop = 0
```

```
for n in numbers:
```

```
    if n % 2 == 0:
```

```
        result_loop += n ** 2
```

```
print(f"Цикл: {result_loop}")
```

```
# 2. map + filter + sum
```

```
result_func = sum(map(lambda x: x ** 2, filter(lambda x: x % 2 == 0, numbers)))
```

```
print(f"map/filter/sum: {result_func}")
```

```
# 3. reduce
```

```
result_reduce = reduce(
```

```
    lambda acc, x: acc + x ** 2 if x % 2 == 0 else acc,
```

```

    numbers,
    0
)
print(f"reduce: {result_reduce}")

```

4. List comprehension (найбільш "пітонічний")

```

result_comp = sum([x ** 2 for x in numbers if x % 2 == 0])
print(f"list comprehension: {result_comp}")

```

Коли що використовувати?

Операція	Кращий підхід
Просте перетворення кожного елемента	map() або list comprehension
Фільтрація елементів	filter() або list comprehension
Складне перетворення з умовою	List comprehension
Зведення до одного значення	reduce() або вбудовані sum(), min(), max()
Ланцюжок операцій	List comprehension або комбінація map/filter

Практичний приклад: обробка даних про студентів

```

from functools import reduce

```

Дані про студентів

```

students = [
    {"name": "Олена Петренко", "grades": [85, 90, 78, 92], "course": 2},
    {"name": "Іван Коваленко", "grades": [75, 88, 95, 70], "course": 1},
    {"name": "Марія Шевченко", "grades": [95, 92, 88, 96], "course": 3},
    {"name": "Петро Бондаренко", "grades": [60, 75, 80, 65], "course": 2},
    {"name": "Анна Кравченко", "grades": [100, 98, 95, 99], "course": 1}
]

```

```
print("📊 АНАЛІЗ ДАНИХ СТУДЕНТІВ")
print("=" * 60)
```

1. Обчислюємо середній бал для кожного студента (map)

```
students_with_avg = list(map(
    lambda s: {**s, "avg": sum(s["grades"]) / len(s["grades"])},
    students
))
```

```
print("\n📋 Студенти з середнім балом:")
```

```
for s in students_with_avg[:3]:
    print(f" {s['name']}: {s['avg']:.2f}")
```

2. Фільтруємо відмінників (avg >= 90) (filter)

```
excellent = list(filter(lambda s: s.get("avg", 0) >= 90, students_with_avg))
```

```
print(f"\n🏆 Відмінників: {len(excellent)}")
```

```
for s in excellent:
```

```
    print(f" {s['name']}")
```

3. Знаходимо загальну суму всіх оцінок (reduce)

```
total_sum = reduce(
    lambda acc, s: acc + sum(s["grades"]),
    students,
    0
)
```

```
print(f"\n📈 Загальна сума всіх оцінок: {total_sum}")
```

4. Групуємо студентів за курсами (reduce)

```
by_course = reduce(
    lambda acc, s: {**acc, s["course"]: acc.get(s["course"], []) + [s["name"]]},
    students,
    {}
)
```

```
print("\n📁 Розподіл за курсами:")
```

```
for course, names in by_course.items():
    print(f" Курс {course}: {', '.join(names)}")
```

5. Найвища оцінка серед усіх студентів (reduce + map)

```
all_grades = reduce(lambda acc, s: acc + s["grades"], students, [])
max_grade = reduce(lambda acc, g: acc if acc > g else g, all_grades)
print(f"\n100 Найвища оцінка: {max_grade}")
```

6. Середній бал групи (map + reduce)

```
avg_grades = list(map(lambda s: sum(s["grades"]) / len(s["grades"]), students))
group_avg = reduce(lambda acc, avg: acc + avg, avg_grades) / len(avg_grades)
print(f"\n111 Середній бал групи: {group_avg:.2f}")
```

4. Рекурсія: функція, що викликає саму себе

Рекурсія – це метод розв’язання задач, при якому функція викликає саму себе. Це потужна техніка, яка особливо корисна для задач, що мають рекурсивну природу: обхід дерев, обчислення факторіалів, чисел Фібоначчі, пошук у лабіринтах тощо.

Базова структура рекурсивної функції

Кожна рекурсивна функція повинна мати дві обов’язкові частини:

1. **Базовий випадок (base case)** – умова, за якої функція не викликає себе, а повертає конкретне значення.
2. **Рекурсивний випадок (recursive case)** – функція викликає себе зі зміненими параметрами, наближаючись до базового випадку.

```
def recursive_function(parameters):
    if base_case_condition:
        return base_value
    else:
        # змінюємо параметри, наближаючись до базового випадку
        return recursive_function(modified_parameters)
```

Приклад 1: Факторіал числа

Факторіал числа n ($n!$) – це добуток всіх чисел від 1 до n . Математично:

- $0! = 1$ (базовий випадок)
- $n! = n * (n-1)!$ (рекурсивний випадок)

```
def factorial(n):
    """Обчислює факторіал числа n рекурсивно"""
```

```

# Базовий випадок
if n <= 1:
    return 1
# Рекурсивний випадок
return n * factorial(n - 1)

# Демонстрація
for i in range(1, 11):
    print(f'{i}! = {factorial(i)}')

# Як це працює для factorial(5):
# factorial(5) = 5 * factorial(4)
#           = 5 * (4 * factorial(3))
#           = 5 * (4 * (3 * factorial(2)))
#           = 5 * (4 * (3 * (2 * factorial(1))))
#           = 5 * (4 * (3 * (2 * 1)))
#           = 5 * (4 * (3 * 2))
#           = 5 * (4 * 6)
#           = 5 * 24
#           = 120

```

Приклад 2: Числа Фібоначчі

Послідовність Фібоначчі: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

- $F(0) = 0$, $F(1) = 1$ (базові випадки)
- $F(n) = F(n-1) + F(n-2)$ (рекурсивний випадок)

```

def fibonacci(n):
    """Повертає n-те число Фібоначчі (рекурсивно)"""
    # Базові випадки
    if n <= 0:
        return 0
    if n == 1:
        return 1
    # Рекурсивний випадок
    return fibonacci(n - 1) + fibonacci(n - 2)

# Демонстрація
print("Перші 10 чисел Фібоначчі:")

```

```
for i in range(10):  
    print(f"F({i}) = {fibonacci(i)}")
```

УВАГА: це дуже неефективно! Спробуйте fibonacci(40) – і чекайте довго

Чому? Бо багато значень обчислюються багаторазово

Проблема ефективності рекурсії

Проста рекурсія для Фібоначчі має експоненційну складність $O(2^n)$. Це означає, що fibonacci(40) вимагатиме мільярдів викликів!

```
import time
```

```
def fibonacci_recursive(n):  
    if n <= 0:  
        return 0  
    if n == 1:  
        return 1  
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
```

```
def fibonacci_iterative(n):  
    """Ітеративна версія (набагато швидша)"""  
    if n <= 0:  
        return 0  
    if n == 1:  
        return 1  
  
    a, b = 0, 1  
    for _ in range(2, n + 1):  
        a, b = b, a + b  
    return b
```

Порівняння швидкості

```
n = 35
```

```
start = time.time()  
result_rec = fibonacci_recursive(n)  
end = time.time()
```

```
print(f"Рекурсивно: F({n}) = {result_rec}, час: {end - start:.2f} сек")
```

```
start = time.time()
```

```
result_iter = fibonacci_iterative(n)
```

```
end = time.time()
```

```
print(f"Ітеративно: F({n}) = {result_iter}, час: {end - start:.6f} сек")
```

Покращення: мемоізація (кешування)

Щоб зробити рекурсію ефективною, можна зберігати вже обчислені значення:

```
def fibonacci_memo(n, memo={}):
```

```
    """Числа Фібоначчі з мемоізацією (кешуванням)"""
```

```
    if n in memo:
```

```
        return memo[n]
```

```
    if n <= 0:
```

```
        return 0
```

```
    if n == 1:
```

```
        return 1
```

```
    memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)
```

```
    return memo[n]
```

Тепер навіть великі значення обчислюються швидко

```
for n in [10, 20, 30, 40]:
```

```
    start = time.time()
```

```
    result = fibonacci_memo(n)
```

```
    end = time.time()
```

```
    print(f"F({n}) = {result}, час: {end - start:.6f} сек")
```

Приклад 3: Обхід вкладених структур

Рекурсія ідеально підходить для роботи з вкладеними структурами даних:

```
def sum_nested_list(nested_list):
```

```
    """Рекурсивно обчислює суму всіх чисел у вкладеному списку"""
```

```
    total = 0
```

```
    for element in nested_list:
```



```

    if isinstance(element, list):
        # Якщо елемент – список, рекурсивно обчислюємо його суму
        total += sum_nested_list(element)
    else:
        # Якщо елемент – число, додаємо його
        total += element
    return total

# Приклад
data = [1, 2, [3, 4, [5, 6], 7], 8, [9, 10]]
print(f"Сума всіх чисел: {sum_nested_list(data)}") # 55

def flatten_list(nested_list):
    """Рекурсивно розгортає вкладений список у плоский"""
    result = []
    for element in nested_list:
        if isinstance(element, list):
            result.extend(flatten_list(element))
        else:
            result.append(element)
    return result

print(f"Плоский список: {flatten_list(data)}") # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```

Приклад 4: Бінарний пошук (рекурсивна версія)

```

def binary_search_recursive(arr, target, left, right):
    """
    Рекурсивний бінарний пошук у відсортованому масиві.
    Повертає індекс елемента або -1, якщо не знайдено.
    """

    # Базовий випадок: елемент не знайдено
    if left > right:
        return -1

    # Знаходимо середину
    mid = (left + right) // 2

```

```

# Базовий випадок: знайшли елемент
if arr[mid] == target:
    return mid

# Рекурсивні випадки
if arr[mid] > target:
    return binary_search_recursive(arr, target, left, mid - 1)
else:
    return binary_search_recursive(arr, target, mid + 1, right)

# Демонстрація
numbers = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
targets = [7, 8, 15]

for target in targets:
    index = binary_search_recursive(numbers, target, 0, len(numbers) - 1)
    if index != -1:
        print(f"Елемент {target} знайдено на позиції {index}")
    else:
        print(f"Елемент {target} не знайдено")

```

Приклад 5: Обхід дерева (файлової системи)

```

import os

def list_files_recursive(path, indent=0):
    """
    Рекурсивно виводить структуру файлів та папок.
    Спрощена імітація без реального os.walk.
    """
    # Імітуємо структуру даних
    structure = {
        "documents": {
            "work": ["report.docx", "presentation.pptx"],
            "personal": ["letter.txt", "photo.jpg"],
            "projects": {
                "python": ["main.py", "utils.py"],
                "web": ["index.html", "style.css"]
            }
        }
    }

```

```

    }
},
"photos": ["vacation.jpg", "family.png"],
"music": ["song1.mp3", "song2.mp3"]
}

```

```

def traverse(current, level):
    for name, content in current.items():
        print(" " * level + f"📁 {name}/")
        if isinstance(content, dict):
            traverse(content, level + 1)
        elif isinstance(content, list):
            for file in content:
                print(" " * (level + 1) + f"📄 {file}")

print("🗄️ ФАЙЛОВА СИСТЕМА:")
traverse(structure, 0)

```

```
list_files_recursive(".")
```

Проблема переповнення стеку

Кожен рекурсивний виклик зберігається в стеку викликів. Якщо рекурсія занадто глибока, стек може переповнитися:

```

def infinite_recursion(n):
    """Демонстрація переповнення стеку"""
    print(f"Виклик {n}")
    return infinite_recursion(n + 1)

```

Розкоментуйте на свій страх і ризик!

infinite_recursion(1) # RecursionError: maximum recursion depth exceeded

Глибину рекурсії можна збільшити, але це не вирішує проблему

```
import sys
```

```
print(f"Поточна глибина рекурсії: {sys.getrecursionlimit()}")
```

sys.setrecursionlimit(10000) # збільшуємо ліміт

Переваги та недоліки рекурсії

Переваги	Недоліки
Код часто елегантніший та зрозуміліший	Може бути менш ефективною (витрати на виклики)
Природно відповідає рекурсивним структурам даних	Ризик переповнення стеку
Легше реалізувати складні алгоритми (обхід дерев)	Складніше налагоджувати
Менше коду для деяких задач	Не всі задачі можна вирішити рекурсивно

Коли використовувати рекурсію?

- Робота з деревоподібними структурами (файлова система, DOM, XML/HTML)
- Алгоритми "розділяй і володарюй" (швидке сортування, злиття)
- Задачі, що мають рекурсивну математичну природу (фрактали)
- Обхід графів (пошук у глибину)
- Коли ітеративне рішення набагато складніше

Практичний приклад: Гра "Ханойські вежі"

```
def hanoi_tower(n, source, target, auxiliary):
```

```
    """
```

Розв'язує головоломку "Ханойські вежі" рекурсивно.

n: кількість дисків

source: початковий стрижень

target: цільовий стрижень

auxiliary: допоміжний стрижень

```
    """
```

```
    if n == 1:
```

```
        print(f"Перемістити диск 1 з {source} на {target}")
```

```
        return
```

```
    # Переміщуємо n-1 дисків з source на auxiliary
```

```
    hanoi_tower(n - 1, source, auxiliary, target)
```

```

# Переміщуємо найбільший диск з source на target
print(f'Перемістити диск {n} з {source} на {target}')

# Переміщуємо n-1 дисків з auxiliary на target
hanoi_tower(n - 1, auxiliary, target, source)

# Розв'язок для 3 дисків
print("РОЗВ'ЯЗОК ХАНОЙСЬКИХ ВЕЖ (3 ДИСКИ):")
hanoi_tower(3, "A", "C", "B")
print(f'Всього ходів: {2**3 - 1}')

```

Підсумки

- Функції вищого порядку (map, filter, reduce) дозволяють писати чистий, декларативний код для обробки колекцій.
- Рекурсія – потужна техніка для задач, які мають рекурсивну природу, але потребує обережності через ризик переповнення стеку та неефективності.
- Мемоізація може значно покращити ефективність рекурсивних функцій.
- Вибір між рекурсією та ітерацією залежить від конкретної задачі та особистих уподобань.

У наступній лекції ми перейдемо до модулів, пакетів та роботи з файлами – це дозволить нам організовувати великі проекти та працювати з даними на диску.

Питання для обговорення

1. Поясніть призначення *args та **kwargs. У яких ситуаціях вони незамінні? Наведіть приклад функції, яка може приймати будь-яку кількість позиційних аргументів.
2. Як розпакувати список у позиційні аргументи при виклику функції? Як розпакувати словник в іменовані аргументи? Наведіть приклади.
3. Проаналізуйте код:

```

def func(a, b, c):
    print(a, b, c)

args = [1, 2, 3]
kwargs = {'a': 1, 'b': 2, 'c': 3}

```

Як викликати функцію func з використанням args та kwargs?

4. Що таке лямбда-функція? Чим вона відрізняється від звичайної функції, створеної через def? У яких ситуаціях лямбди особливо зручні?

5. Наведіть приклад використання лямбда-функції як ключа для сортування списку словників. Наприклад, відсортуйте список студентів за віком.

6. Поясніть роботу функції map(). Що вона повертає? Як перетворити результат map() у список? Наведіть приклад перетворення списку чисел у список їх квадратів за допомогою map() та лямбда-функції.

7. Поясніть роботу функції filter(). Наведіть приклад фільтрації списку чисел для отримання тільки парних чисел.

8. Що таке рекурсія? З яких двох обов'язкових частин складається будь-яка рекурсивна функція? Наведіть приклад рекурсивної функції для обчислення факторіала.

9. Проаналізуйте рекурсивну функцію Фібоначчі:

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

Чому ця функція дуже неефективна для великих n? Як можна покращити її продуктивність?

10. У чому небезпека глибокої рекурсії? Що таке переповнення стеку (stack overflow) і як його уникнути? Як в Python можна дізнатися максимальну глибину рекурсії?

ЛЕКЦІЯ 11

Модулі, пакети та робота з файлами

План лекції

1. Модулі та імпорт: організація коду в окремі файли.

Пояснимо, що таке модуль, як створювати власні модулі, як імпортувати їх за допомогою `import` та `from...import`. Розберемо різницю між різними способами імпорту, використання псевдонімів (`as`) і чому це важливо для уникнення конфліктів імен.

2. Пакети та структура проекту: `init.py` та `if name == "main"`.

Покажемо, як організовувати великі проекти за допомогою пакетів. Розберемо призначення файлу `__init__.py`. Пояснимо одну з найважливіших конструкцій – `if __name__ == "__main__"`, яка дозволяє використовувати файл і як модуль, і як самостійну програму.

3. Огляд стандартної бібліотеки: `math`, `random`, `datetime`, `os`, `sys`.

Познайомимось з найкориснішими модулями стандартної бібліотеки Python. На практичних прикладах покажемо, як використовувати математичні функції, генерувати випадкові числа, працювати з датами та часом, взаємодіяти з операційною системою.

4. Робота з файлами: читання, запис, контекстний менеджер `with`.

Детально розберемо роботу з файлами: відкриття в різних режимах, читання та запис тексту, роботу з шляхами за допомогою `pathlib`. Особливу увагу приділимо конструкції `with` – безпечному та сучасному способу роботи з файлами, який гарантує їх правильне закриття.

1. Модулі та імпорт: організація коду в окремі файли

Коли наші програми стають більшими, тримати весь код в одному файлі стає незручно. Код важко читати, важко підтримувати, важко тестувати. Крім того, ми часто хочемо використовувати код, написаний іншими програмістами. Саме для цього в Python існують **модулі**.

Що таке модуль?

Модуль – це просто файл з розширенням `.py`, який містить код Python (функції, класи, змінні). Модулі дозволяють логічно організовувати код і повторно використовувати його в різних програмах.

Створення власного модуля

Створимо наш перший модуль. Назвемо його `geometry.py`:

```
# geometry.py
```

```
"""
```

Модуль для геометричних обчислень.

Містить функції для роботи з різними фігурами.

```
"""
```

```
import math
```

```
def circle_area(radius):
```

```
    """Обчислює площу круга"""
```

```
    return math.pi * radius ** 2
```

```
def circle_circumference(radius):
```

```
    """Обчислює довжину кола"""
```

```
    return 2 * math.pi * radius
```

```
def rectangle_area(length, width):
```

```
    """Обчислює площу прямокутника"""
```

```
    return length * width
```

```
def rectangle_perimeter(length, width):
```

```
    """Обчислює периметр прямокутника"""
```

```
    return 2 * (length + width)
```

```
def triangle_area(base, height):
```

```
    """Обчислює площу трикутника за основою та висотою"""
```

```
    return 0.5 * base * height
```

```
# Константи
```

```
PI = math.pi
```

```
DEFAULT_PRECISION = 2
```

Тепер створимо інший файл, скажімо main.py, у тій самій папці, і навчимося імпортувати наш модуль.

Способи імпорту модулів

1. Імпорт всього модуля:


```
# main.py
import geometry

# Використовуємо функції з модуля через крапку
radius = 5
area = geometry.circle_area(radius)
print(f'Площа круга: {area:.{geometry.DEFAULT_PRECISION}f}')
print(f'Значення PI з модуля: {geometry.PI}')
```

2. Імпорт конкретних функцій:

```
# main.py
from geometry import circle_area, rectangle_area

# Можна використовувати без крапки
print(f'Площа круга: {circle_area(5)}')
print(f'Площа прямокутника: {rectangle_area(10, 20)}')
```

```
# Але функція triangle_area не імпортована – помилка!
# print(triangle_area(10, 5)) # NameError
```

3. Імпорт з псевдонімом (alias):

```
# main.py
import geometry as geo
from geometry import circle_area as ca, PI

print(f'Площа круга: {ca(5)}')
print(f'Довжина кола: {geo.circle_circumference(5)}')
print(f'PI: {PI}')
```

4. Імпорт всього вмісту (не рекомендується!):

```
# main.py
from geometry import *

# Тепер всі функції доступні без префікса
print(circle_area(5))
print(rectangle_perimeter(10, 20))
```

АЛЕ! Це погана практика, особливо у великих проектах:
- Можливі конфлікти імен
- Неясно, звідки взялася функція
- Важко читати код

Чому `import *` – це погано?

Нехай є два модулі:

module1.py

```
def print_info():  
    print("Інформація з module1")
```

module2.py

```
def print_info():  
    print("Інформація з module2")
```

main.py

```
from module1 import *  
from module2 import *
```

`print_info()` *# Яка функція виконається? Остання імпортована!*

Це призводить до непередбачуваної поведінки

Пошук модулів

Коли Python зустрічає `import geometry`, він шукає модуль у таких місцях:

1. Поточна директорія
2. Змінна середовища `PYTHONPATH`
3. Стандартні шляхи встановлення Python

Подивитися всі шляхи пошуку можна так:

```
import sys  
print(sys.path)
```

Практичний приклад: створення бібліотеки корисних функцій

Створимо більш складний модуль `string_utils.py`:

string_utils.py

```
"""
```

Модуль для роботи з рядками.

Містить корисні функції для обробки тексту.

"""

```
def reverse_string(s):
```

```
    """Повертає перевернутий рядок"""
```

```
    return s[::-1]
```

```
def count_vowels(s):
```

```
    """Підраховує кількість голосних у рядку"""
```

```
    vowels = "аеіоіуяюєїАЕІОІУЯЮЄЇ"
```

```
    return sum(1 for char in s if char in vowels)
```

```
def is_palindrome(s):
```

```
    """Перевіряє, чи є рядок паліндромом"""
```

```
    # Видаляємо пробіли та переводимо в нижній регістр
```

```
    cleaned = ''.join(s.lower().split())
```

```
    return cleaned == cleaned[::-1]
```

```
def capitalize_words(s):
```

```
    """Робить великою першу літеру кожного слова"""
```

```
    return ' '.join(word.capitalize() for word in s.split())
```

```
def truncate(s, max_length, suffix="..."):
```

```
    """Обрізає рядок до заданої довжини з додаванням суфікса"""
```

```
    if len(s) <= max_length:
```

```
        return s
```

```
    return s[:max_length - len(suffix)] + suffix
```

```
# Константи
```

```
VOWELS = "аеіоіуяюєї"
```

```
CONSONANTS = "бвггджзйклмнпрстфхцчшщ"
```

Тепер створимо програму, яка використовує наш модуль:

```
# text_analyzer.py
```

```
import string_utils as su
```

```
def analyze_text(text):
```

```

"""Аналізує текст за допомогою функцій з модуля"""
print("■ АНАЛІЗ ТЕКСТУ")
print("=" * 50)
print(f"Оригінальний текст: {su.truncate(text, 50)}")
print(f"Перевернутий текст: {su.truncate(su.reverse_string(text), 50)}")
print(f"Кількість голосних: {su.count_vowels(text)}")
print(f"Кількість приголосних: {sum(1 for c in text if c in
su.CONSONANTS)}")
print(f"Чи є паліндромом? {'Так' if su.is_palindrome(text) else 'Hi'}")
print(f"З великими літерами: {su.capitalize_words(text)}")
print("=" * 50)

# Тестуємо
analyze_text("радар")
print()
analyze_text("Python - це потужна мова програмування")

```

Модуль як виконуваний файл

Кожен модуль має спеціальну змінну `__name__`. Якщо модуль запускається безпосередньо, `__name__` дорівнює `"__main__"`. Якщо модуль імпортується, `__name__` дорівнює назві модуля. Це дозволяє створювати модулі, які можна і запускати самостійно, і імпортувати.

Додамо в кінець `string_utils.py`:

```

# string_utils.py (додаємо в кінець)

if __name__ == "__main__":
    # Цей код виконається, тільки якщо файл запущено напряму
    print("Тестування модуля string_utils:")
    print(reverse_string("Python"))
    print(count_vowels("Hello World"))
    print(is_palindrome("радар"))
    print(capitalize_words("python це чудово"))

```

Тепер, якщо ми запустимо `python string_utils.py`, побачимо тести. Якщо імпортуємо модуль в іншій програмі – тести не виконуються.

2. Пакети та структура проекту: `init.py` та `if name == "main"`

Коли модулів стає багато, їх зручно організовувати в **пакети** – каталоги з модулями. Пакети дозволяють створювати ієрархічну структуру проекту.

Створення пакету

Створимо структуру пакету для математичних операцій:

```
math_operations/  
  __init__.py  
  basic.py  
  advanced.py  
  geometry/  
    __init__.py  
    circle.py  
    rectangle.py  
    triangle.py
```

Кожен каталог, який є пакетом, повинен містити файл `__init__.py` (може бути порожнім, але обов'язково повинен існувати в Python до версії 3.3; у сучасних версіях він не обов'язковий, але рекомендується).

Файл `init.py`

Файл `__init__.py` може бути порожнім, але часто в ньому:

- Ініціалізуються змінні пакету
- Імпортуються ключові функції для зручності
- Визначається, що імпортується при `from package import *`

Наприклад, `math_operations/__init__.py`:

```
# math_operations/__init__.py  
"""
```

Пакет для математичних операцій.

Містить базові та просунуті математичні функції.

```
"""
```

```
# Імпортуємо ключові функції для зручного доступу
```

```
from .basic import add, subtract, multiply, divide
```

```
from .advanced import power, sqrt, factorial
```

```
# Що імпортується при from math_operations import *  
__all__ = ['add', 'subtract', 'multiply', 'divide', 'power', 'sqrt', 'factorial']
```

```
print("Пакет math_operations завантажено") # Виконається при імпорті пакету
```

Модуль basic.py:

```
# math_operations/basic.py  
"""Базові арифметичні операції"""
```

```
def add(a, b):  
    """Повертає суму a + b"""  
    return a + b
```

```
def subtract(a, b):  
    """Повертає різницю a - b"""  
    return a - b
```

```
def multiply(a, b):  
    """Повертає добуток a * b"""  
    return a * b
```

```
def divide(a, b):  
    """Повертає частку a / b"""  
    if b == 0:  
        raise ValueError("Ділення на нуль!")  
    return a / b
```

Модуль advanced.py:

```
# math_operations/advanced.py  
"""Просунуті математичні функції"""  
import math
```

```
def power(x, n):  
    """Повертає x в степені n"""  
    return x ** n
```

```
def sqrt(x):
    """Повертає квадратний корінь з x"""
    if x < 0:
        raise ValueError("Корінь з від'ємного числа")
    return math.sqrt(x)

def factorial(n):
    """Повертає факторіал числа n"""
    if n < 0:
        raise ValueError("Факторіал від'ємного числа не визначено")
    if n <= 1:
        return 1
    return n * factorial(n - 1)
```

Підпакети geometry:

math_operations/geometry/__init__.py:

```
# math_operations/geometry/__init__.py
"""Підпакет для геометричних обчислень"""
from .circle import circle_area, circle_circumference
from .rectangle import rectangle_area, rectangle_perimeter
from .triangle import triangle_area

__all__ = [
    'circle_area', 'circle_circumference',
    'rectangle_area', 'rectangle_perimeter',
    'triangle_area'
]
```

math_operations/geometry/circle.py:

```
# math_operations/geometry/circle.py
import math

def circle_area(radius):
    """Площа круга"""
    return math.pi * radius ** 2
```

```
def circle_circumference(radius):
    """Довжина кола"""
    return 2 * math.pi * radius
math_operations/geometry/rectangle.py:
```

```
# math_operations/geometry/rectangle.py
```

```
def rectangle_area(length, width):
    """Площа прямокутника"""
    return length * width
```

```
def rectangle_perimeter(length, width):
    """Периметр прямокутника"""
    return 2 * (length + width)
```

```
math_operations/geometry/triangle.py:
```

```
# math_operations/geometry/triangle.py
```

```
def triangle_area(base, height):
    """Площа трикутника за основою та висотою"""
    return 0.5 * base * height
```

Використання пакету

Тепер створимо програму, яка використовує наш пакет різними способами:

```
# calculator.py
```

```
# Спосіб 1: імпорт всього пакету
```

```
import math_operations as mo
```

```
print("=== БАЗОВІ ОПЕРАЦІЇ ===")
print(f"5 + 3 = {mo.add(5, 3)}")
print(f"10 - 4 = {mo.subtract(10, 4)}")
print(f"6 * 7 = {mo.multiply(6, 7)}")
print(f"15 / 3 = {mo.divide(15, 3)}")
```

```
print("\n=== ПРОСУНУТІ ОПЕРАЦІЇ ===")
print(f"2^8 = {mo.power(2, 8)}")
print(f"√16 = {mo.sqrt(16)}")
print(f"5! = {mo.factorial(5)}")
```


Спосіб 2: імпорт конкретних функцій

```
from math_operations import add, multiply, power
```

```
print(f"\n3 + 7 = {add(3, 7)}")
```

```
print(f"4 * 9 = {multiply(4, 9)}")
```

Спосіб 3: імпорт з підпаketу

```
from math_operations.geometry import circle_area, rectangle_area
```

```
print(f"\nПлоща круга (r=5): {circle_area(5):.2f}")
```

```
print(f"Площа прямокутника (4x7): {rectangle_area(4, 7)}")
```

Спосіб 4: імпорт всього підпаketу з псевдонімом

```
from math_operations.geometry import triangle as t
```

```
print(f"Площа трикутника (10x5): {t.triangle_area(10, 5)}")
```

Спосіб 5: імпорт всього, що в __all__

```
from math_operations import *
```

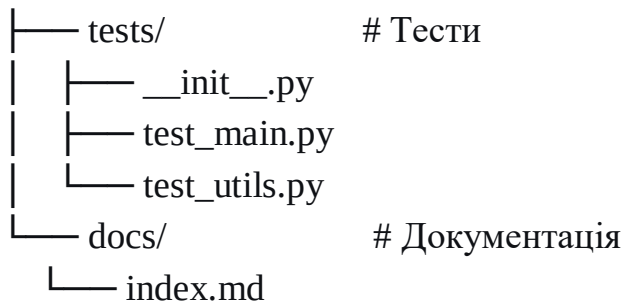
```
print(f"\nВіднімання: {subtract(100, 45)}")
```

```
print(f"Факторіал: {factorial(6)}")
```

Структура типового Python-проєкту

Коли ви створюєте справжній проєкт, структура зазвичай виглядає так:

```
my_project/
├── README.md          # Опис проєкту
├── requirements.txt    # Залежності (потрібні бібліотеки)
├── setup.py           # Файл встановлення (для бібліотек)
├── .gitignore         # Файли, які не треба комітити в git
├── my_project/        # Основний паке́т проєкту
│   ├── __init__.py
│   ├── main.py        # Головний файл програми
│   ├── utils/         # Підпаке́ти
│   │   ├── __init__.py
│   │   ├── helpers.py
│   │   └── validators.py
│   └── models/        # Інший підпаке́т
│       ├── __init__.py
│       └── user.py
```



if name == "main" – детальніше

Ця конструкція надзвичайно корисна. Вона дозволяє файлу бути одночасно і бібліотекою (яку можна імпортувати), і виконуваною програмою.

Створимо файл `useful_functions.py`:

```
# useful_functions.py
```

```
def greet(name):
```

```
    """Повертає вітання"""
```

```
    return f"Привіт, {name}!"
```

```
def add(a, b):
```

```
    """Додає два числа"""
```

```
    return a + b
```

```
def main():
```

```
    """Функція, яка виконується при запуску файлу напряду"""
```

```
    print("=== ТЕСТУВАННЯ МОДУЛЯ ===")
```

```
    print(greet("Олена"))
```

```
    print(f"5 + 3 = {add(5, 3)}")
```

```
# Демонстрація інших функцій
```

```
    name = input("Як вас звати? ")
```

```
    print(greet(name))
```

```
    x = float(input("Введіть перше число: "))
```

```
    y = float(input("Введіть друге число: "))
```

```
    print(f"{x} + {y} = {add(x, y)}")
```

```
if __name__ == "__main__":
```

```
# Цей код виконається, тільки якщо файл запущено напряду
```

```
    main()
```

```
print("\nМодуль виконано як програму")
else:
    # Цей код виконається, якщо файл імпортовано
    print(f"Модуль {__name__} імпортовано")
Тепер створимо app.py, який імпортує цей модуль:
```

```
# app.py
import useful_functions as uf

print("Програма app.py використовує useful_functions як модуль")
print(uf.greet("Іван"))
print(f"10 + 20 = {uf.add(10, 20)}")

# main() з useful_functions не виконається автоматично
# Але ми можемо викликати її явно, якщо хочемо
# uf.main() # розкоментуйте, щоб побачити
```

Переваги такої структури:

1. **Тестування:** Можна додати тести прямо в модуль і запускати їх при розробці.
2. **Документація:** Можна показувати приклади використання.
3. **Гнучкість:** Один файл працює і як бібліотека, і як програма.
4. **Профілювання:** Можна додати код для вимірювання продуктивності.

Практичний приклад: створення CLI-утиліти

Створимо корисну утиліту для роботи з файлами, яка буде і модулем, і самостійною програмою:

```
# file_utils.py
"""
Модуль для роботи з файлами.
Може використовуватися як бібліотека або як CLI-утиліта.
"""

import os
import sys

def get_file_size(filename):
    """Повертає розмір файлу в байтах"""
```

```

try:
    return os.path.getsize(filename)
except FileNotFoundError:
    return None

def count_lines(filename):
    """Підраховує кількість рядків у файлі"""
    try:
        with open(filename, 'r', encoding='utf-8') as f:
            return len(f.readlines())
    except FileNotFoundError:
        return None

def file_exists(filename):
    """Перевіряє, чи існує файл"""
    return os.path.exists(filename)

def get_file_extension(filename):
    """Повертає розширення файлу"""
    return os.path.splitext(filename)[1]

def main():
    """Головна функція для CLI"""
    print("🔧 FILE UTILS - Утиліта для роботи з файлами")
    print("=" * 50)

    if len(sys.argv) < 2:
        print("Використання: python file_utils.py <ім'я_файлу>")
        print("Або: python file_utils.py --help")
        return

    filename = sys.argv[1]

    if filename == "--help":
        print("Команди:")
        print(" python file_utils.py <файл> - показати інформацію про файл")
        print(" python file_utils.py --size <файл> - тільки розмір")

```

```

print(" python file_utils.py --lines <файл> - тільки кількість рядків")
return

if not file_exists(filename):
    print(f"✗ Файл '{filename}' не знайдено")
    return

print(f"📄 Файл: {filename}")
print(f"📏 Розмір: {get_file_size(filename)} байт")
print(f"🏷️ Розширення: {get_file_extension(filename)}")

lines = count_lines(filename)
if lines is not None:
    print(f"📄 Кількість рядків: {lines}")

print("=" * 50)

if __name__ == "__main__":
    main()
else:
    print(f"Модуль {__name__} імпортовано. Використовуйте функції:
get_file_size, count_lines, file_exists, get_file_extension")

```

Тепер ми можемо:

- Використовувати як модуль: `from file_utils import get_file_size`
- Використовувати як програму: `python file_utils.py myfile.txt`

У наступних пунктах ми розглянемо стандартну бібліотеку Python та роботу з файлами більш детально.

3. Огляд стандартної бібліотеки: `math`, `random`, `datetime`, `os`, `sys`

Одна з найбільших переваг Python – його багата стандартна бібліотека. Це набір модулів, які вже включені в Python і готові до використання. Вам не потрібно нічого додатково встановлювати – просто імпортуйте потрібний модуль і користуйтеся. Це економить тисячі годин роботи, адже найпоширеніші задачі вже розв'язані за вас.

Модуль math – математичні функції

Модуль math надає доступ до математичних функцій та констант.

```
import math
```

```
print("MODUL MATH")
```

```
print("=" * 50)
```

Константи

```
print(f"π (пі): {math.pi}")
```

```
print(f"e (основа натурального логарифма): {math.e}")
```

```
print(f"τ (тау, 2π): {math.tau}")
```

Тригонометричні функції

```
angle = math.pi / 4 # 45 градусів
```

```
print(f"\nТригонометрія (кут = 45°):")
```

```
print(f"sin(45°): {math.sin(angle):.4f}")
```

```
print(f"cos(45°): {math.cos(angle):.4f}")
```

```
print(f"tan(45°): {math.tan(angle):.4f}")
```

```
print(f"arcsin(0.7071): {math.asin(0.7071):.4f} рад")
```

Степенева та логарифмічна функції

```
print(f"\nСтепені та логарифми:")
```

```
print(f"2^10: {math.pow(2, 10)}") # 2 в степені 10
```

```
print(f"Квадратний корінь з 16: {math.sqrt(16)}")
```

```
print(f"exp(2) (e²): {math.exp(2):.4f}")
```

```
print(f"log10(1000): {math.log10(1000)}")
```

```
print(f"log2(1024): {math.log2(1024)}")
```

```
print(f"ln(e³): {math.log(math.e**3):.4f}")
```

Округлення

```
print(f"\nОкруглення:")
```

```
print(f"ceil(4.2) (вгору): {math.ceil(4.2)}")
```

```
print(f"floor(4.8) (вниз): {math.floor(4.8)}")
```

```
print(f"trunc(4.8) (відкинути дробову): {math.trunc(4.8)}")
```

Інші корисні функції

```
print(f"\nІнше:")
print(f'Факторіал 5: {math.factorial(5)}')
print(f'НСД(24, 36): {math.gcd(24, 36)}') # Найбільший спільний дільник
print(f'Гіпотенуза (3, 4): {math.hypot(3, 4)}') #  $\sqrt{3^2 + 4^2}$ 
```

Перевірка на спеціальні значення

```
print(f"\nПеревірка чисел:")
print(f'Чи є inf числом? {math.isinf(float('inf'))}')
print(f'Чи є NaN числом? {math.isnan(float('nan'))}')
print(f'Чи близькі 0.1 + 0.2 до 0.3? {math.isclose(0.1 + 0.2, 0.3)}')
```

Модуль random – генерація випадкових чисел

Модуль random використовується для створення випадкових значень – для ігор, тестування, симуляцій тощо.

```
import random
```

```
print("🎲 МОДУЛЬ RANDOM")
print("=" * 50)
```

Випадкове число в діапазоні [0, 1)

```
print(f"random(): {random.random()}")
```

Випадкове ціле число в діапазоні [a, b]

```
print(f"randint(1, 10): {random.randint(1, 10)}")
```

Випадкове число з плаваючою комою в діапазоні [a, b]

```
print(f"uniform(5, 10): {random.uniform(5, 10):.2f}")
```

Випадковий вибір з послідовності

```
fruits = ["яблуко", "банан", "апельсин", "груша", "слива"]
print(f'Вибір зі списку: {random.choice(fruits)}')
```

Кілька випадкових виборів (з поверненням)

```
print(f'Три вибори (з поверненням): {random.choices(fruits, k=3)}')
```

Кілька випадкових виборів (без повернення)

```
print(f'Два унікальні вибори: {random.sample(fruits, 2)}')
```

Перемішування списку

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(numbers)
print(f"Перемішаний список: {numbers}")
```

Встановлення зерна для відтворюваності

```
random.seed(42)
print(f"\n3 однаковим seed (42):")
print(f" {random.randint(1, 100)}")
print(f" {random.randint(1, 100)}")
print(f" {random.randint(1, 100)}")
```

Різні розподіли

```
print(f"\nРізні розподіли:")
print(f"Гауссів розподіл ( $\mu=0$ ,  $\sigma=1$ ): {random.gauss(0, 1):.4f}")
print(f"Експоненційний розподіл ( $\lambda=2$ ): {random.expovariate(2):.4f}")
```

Практичний приклад: гра "Вгадай число"

```
import random

def guess_number_game():
    print("🎮 ВГАДАЙ ЧИСЛО")
    print("=" * 50)
    print("Я загадав число від 1 до 100. Спробуй вгадати!")

    secret = random.randint(1, 100)
    attempts = 0

    while True:
        try:
            guess = int(input("Твій варіант: "))
            attempts += 1

            if guess < secret:
                print("❌ Більше!")
```



```

elif guess > secret:
    print("✗ Менше!")
else:
    print(f"✓ Вітаю! Ти вгадав число {secret} за {attempts} спроб!")
    break
except ValueError:
    print("✗ Будь ласка, введи число")

```

guess_number_game()

Модуль datetime – робота з датою та часом

Модуль datetime дозволяє працювати з датами, часом та інтервалами.

```
import datetime
```

```

print("📅 МОДУЛЬ DATETIME")
print("=" * 50)

```

Поточна дата і час

```

now = datetime.datetime.now()
print(f"Поточний момент: {now}")
print(f"Тип: {type(now)}")

```

Окремі компоненти

```

print(f"\nКомпоненти дати/часу:")
print(f"Рік: {now.year}")
print(f"Місяць: {now.month}")
print(f"День: {now.day}")
print(f"Година: {now.hour}")
print(f"Хвилина: {now.minute}")
print(f"Секунда: {now.second}")
print(f"Мікросекунда: {now.microsecond}")

```

Тільки дата або тільки час

```

today = datetime.date.today()
current_time = datetime.datetime.now().time()
print(f"\nСьогодні: {today}")

```

```
print(f"Поточний час: {current_time}")
```

```
# Створення власної дати
```

```
birthday = datetime.date(2000, 5, 15)
```

```
print(f"\nДата народження: {birthday}")
```

```
print(f"День тижня: {birthday.strftime('%A')}")
```

```
# Форматування дати
```

```
print(f"\nФорматування:")
```

```
print(f" {now.strftime('%d.%m.%Y %H:%M:%S')}")
```

```
print(f" {now.strftime('%A, %d %B %Y')}")
```

```
print(f" {now.strftime('%Y-%m-%d')}")
```

```
# Розбір рядка в дату
```

```
date_str = "2023-12-25"
```

```
parsed_date = datetime.datetime.strptime(date_str, "%Y-%m-%d")
```

```
print(f"\nРозбір рядка '{date_str}': {parsed_date}")
```

```
# Арифметика з датами
```

```
delta = datetime.timedelta(days=7, hours=3)
```

```
future = now + delta
```

```
past = now - delta
```

```
print(f"\nЧерез тиждень: {future}")
```

```
print(f"Тиждень тому: {past}")
```

```
# Різниця між датами
```

```
new_year = datetime.datetime(now.year + 1, 1, 1)
```

```
time_to_new_year = new_year - now
```

```
print(f"\nДо Нового року: {time_to_new_year.days} днів,  
{time_to_new_year.seconds//3600} годин")
```

Модуль os – взаємодія з операційною системою

Модуль os дозволяє виконувати операції, пов'язані з ОС: робота з файловою системою, змінними середовища, запуск процесів тощо.

```
import os
```

```
print(" 📖 МОДУЛЬ OS")
```

```

print("=" * 50)

# Поточна директорія
current_dir = os.getcwd()
print(f"Поточна директорія: {current_dir}")

# Зміна директорії
# os.chdir('/tmp') # розкоментуйте для зміни

# Список файлів у директорії
files = os.listdir('.')
print(f"\nФайли в поточній директорії:")
for f in files[:5]: # перші 5
    print(f" {f}")

# Створення директорії
new_dir = "test_dir"
if not os.path.exists(new_dir):
    os.mkdir(new_dir)
    print(f"\nСтворено директорію: {new_dir}")

# Видалення директорії
if os.path.exists(new_dir):
    os.rmdir(new_dir)
    print(f"Видалено директорію: {new_dir}")

# Інформація про файл
if os.path.exists(__file__): # поточний файл
    file_info = os.stat(__file__)
    print(f"\nІнформація про поточний файл:")
    print(f" Розмір: {file_info.st_size} байт")
    print(f" Час створення:
{datetime.datetime.fromtimestamp(file_info.st_ctime)})")
    print(f" Час модифікації:
{datetime.datetime.fromtimestamp(file_info.st_mtime)})")

# Змінні середовища

```

```
print(f"\nЗмінні середовища:")
print(f" HOME: {os.environ.get('HOME', 'не встановлено')}")
print(f" USER: {os.environ.get('USER', 'не встановлено')}")
print(f" PATH: {os.environ.get('PATH', 'не встановлено')[:50]}...")
```

Шляхи

```
path = os.path.join('folder', 'subfolder', 'file.txt')
print(f"\nОб'єднання шляхів: {path}")
```

```
filename = '/home/user/docs/file.txt'
print(f"Батьківська директорія: {os.path.dirname(filename)}")
print(f"Ім'я файлу: {os.path.basename(filename)}")
print(f"Розширення: {os.path.splitext(filename)[1]}")
```

Модуль sys – системні параметри

Модуль sys надає доступ до параметрів та функцій, специфічних для інтерпретатора Python.

```
import sys
```

```
print("❖ МОДУЛЬ SYS")
print("=" * 50)
```

Версія Python

```
print(f"Версія Python: {sys.version}")
print(f"Версія (коротко): {sys.version_info.major}.{sys.version_info.minor}")
```

Аргументи командного рядка

```
print(f"\nАргументи командного рядка: {sys.argv}")
print(f" Ім'я скрипта: {sys.argv[0]}")
if len(sys.argv) > 1:
    print(f" Перший аргумент: {sys.argv[1]}")
```

Шляхи пошуку модулів

```
print(f"\nШляхи пошуку модулів:")
for i, path in enumerate(sys.path[:5]): # перші 5
    print(f" {i+1}. {path}")
```

```

# Платформа
print(f"\nПлатформа: {sys.platform}")

# Вихід з програми
# sys.exit(0) # розкоментуйте для виходу

# Інформація про пам'ять
print(f"\nРозмір цілого числа: {sys.getsizeof(42)} байт")
print(f"Розмір списку з 10 елементів: {sys.getsizeof([1,2,3,4,5,6,7,8,9,10])} байт")
print(f"Розмір рядка: {sys.getsizeof('Hello, World!')} байт")

# Максимальна глибина рекурсії
print(f"\nМаксимальна глибина рекурсії: {sys.getrecursionlimit()}")

# Перенаправлення виведення
# sys.stdout.write("Це виведено через sys.stdout.write()\n")

```

Практичний приклад: створення резервної копії

```

import os
import datetime
import shutil

def create_backup(source_dir, backup_dir=None):
    """
    Створює резервну копію директорії.
    """
    print("📁 СТВОРЕННЯ РЕЗЕРВНОЇ КОПІЇ")
    print("=" * 60)

    # Перевіряємо, чи існує джерело
    if not os.path.exists(source_dir):
        print(f"❌ Директорія {source_dir} не існує")
        return False

    # Якщо не вказано директорію для бекапу, створюємо ім'я з датою

```

```

if backup_dir is None:
    timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
    backup_dir = f"backup_{timestamp}"

try:
    # Створюємо резервну копію
    shutil.copytree(source_dir, backup_dir)
    print(f"✓ Резервну копію створено: {backup_dir}")

    # Інформація про бекап
    size = sum(os.path.getsize(os.path.join(dirpath, filename))
               for dirpath, _, filenames in os.walk(backup_dir)
               for filename in filenames)

    print(f"📏 Розмір бекапу: {size} байт ({size/1024:.2f} КБ)")
    print(f"🕒 Час створення: {datetime.datetime.now()}")

    return True

except Exception as e:
    print(f"✗ Помилка при створенні бекапу: {e}")
    return False

# Приклад використання
# create_backup(".", "my_backup")

```

4. Робота з файлами: читання, запис, контекстний менеджер with

Робота з файлами – одна з найважливіших навичок у програмуванні. Вона дозволяє зберігати дані між запусками програм, обробляти великі обсяги інформації, створювати звіти та багато іншого.

Відкриття файлів: функція open()

Для роботи з файлом спочатку потрібно його відкрити за допомогою функції open():

```
file = open("example.txt", "r") # відкрити для читання
# ... робота з файлом
file.close() # обов'язково закрити!
```

Режими відкриття файлів:

Режим	Опис
'r'	Читання (за замовчуванням). Файл має існувати
'w'	Запис. Якщо файл існує, його вміст стирається
'a'	Додавання. Запис у кінець файлу
'x'	Ексклюзивне створення. Помилка, якщо файл існує
'b'	Бінарний режим (додається до інших режимів)
't'	Текстовий режим (за замовчуванням)
'+'	Читання та запис (додається до інших режимів)

Контекстний менеджер with

Найкращий спосіб роботи з файлами – використовувати конструкцію with. Вона автоматично закриває файл після завершення блоку, навіть якщо сталася помилка.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# Файл автоматично закривається тут
```

Читання з файлів

```
# Створимо тестовий файл для прикладів
with open("sample.txt", "w", encoding="utf-8") as f:
    f.write("Перший рядок\n")
    f.write("Другий рядок\n")
    f.write("Третій рядок\n")
    f.write("Четвертий рядок\n")
```

```
f.write("П'ятий рядок\n")
```

```
print("📖 ЧИТАННЯ З ФАЙЛІВ")
```

```
print("=" * 60)
```

Спосіб 1: read() – все одразу

```
with open("sample.txt", "r", encoding="utf-8") as f:
    content = f.read()
    print("read():")
    print(repr(content))
    print()
```

Спосіб 2: readline() – по одному рядку

```
with open("sample.txt", "r", encoding="utf-8") as f:
    print("readline():")
    line = f.readline()
    while line:
        print(f" {repr(line)}")
        line = f.readline()
    print()
```

Спосіб 3: readlines() – всі рядки в список

```
with open("sample.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()
    print(f"readlines(): {lines}")
    print()
```

Спосіб 4: ітерація по файлу (найкращий для великих файлів!)

```
with open("sample.txt", "r", encoding="utf-8") as f:
    print("Ітерація по файлу:")
    for i, line in enumerate(f, 1):
        print(f" Рядок {i}: {line.rstrip()}")
```

Запис у файли

```
print("📝 ЗАПИС У ФАЙЛИ")
```

```
print("=" * 60)
```


Режим 'w' – перезапис

```
with open("output.txt", "w", encoding="utf-8") as f:
    f.write("Це перший рядок\n")
    f.write("Це другий рядок\n")
    f.write("Це третій рядок\n")
print("Файл output.txt створено з 3 рядками")
```

Режим 'a' – додавання в кінець

```
with open("output.txt", "a", encoding="utf-8") as f:
    f.write("Це четвертий рядок (доданий)\n")
    f.write("Це п'ятий рядок (доданий)\n")
print("Додано ще 2 рядки")
```

writelines() – запис списку рядків

```
lines = ["Рядок A\n", "Рядок B\n", "Рядок C\n"]
with open("output2.txt", "w", encoding="utf-8") as f:
    f.writelines(lines)
print("Файл output2.txt створено за допомогою writelines()")
```

Перевіримо результат

```
with open("output.txt", "r", encoding="utf-8") as f:
    print("\nВміст output.txt:")
    print(f.read())
```

Робота з бінарними файлами

```
print("\n📁 БІНАРНІ ФАЙЛИ")
print("=" * 60)
```

Запис бінарних даних

```
data = bytes([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
with open("binary.dat", "wb") as f:
    f.write(data)
print("Бінарний файл binary.dat створено")
```

Читання бінарних даних

```
with open("binary.dat", "rb") as f:
```

```
read_data = f.read()
print(f'Прочитано {len(read_data)} байт: {read_data}')
```

Робота з JSON

JSON – найпопулярніший формат для обміну даними. Модуль json входить до стандартної бібліотеки.

```
import json

print("\n📁 РОБОТА З JSON")
print("=" * 60)

# Дані для збереження
data = {
    "name": "Олена Петренко",
    "age": 19,
    "city": "Київ",
    "grades": [90, 85, 92, 88],
    "is_student": True,
    "courses": ["Python", "Математика", "Фізика"]
}

# Запис у JSON-файл
with open("data.json", "w", encoding="utf-8") as f:
    json.dump(data, f, ensure_ascii=False, indent=4)
print("JSON-файл створено")

# Читання з JSON-файлу
with open("data.json", "r", encoding="utf-8") as f:
    loaded_data = json.load(f)
    print("Прочитані дані:")
    print(f" Ім'я: {loaded_data['name']}")
    print(f" Вік: {loaded_data['age']}")
    print(f" Місто: {loaded_data['city']}")
    print(f" Оцінки: {loaded_data['grades']}")
    print(f" Студент: {loaded_data['is_student']}")
```

Робота з CSV

CSV (Comma-Separated Values) – поширений формат для табличних даних.

```
import csv
```

```
print("\n📄 РОБОТА З CSV")
```

```
print("=" * 60)
```

```
# Дані для запису
```

```
students = [  
    ["Ім'я", "Вік", "Місто", "Середній бал"],  
    ["Олена", 19, "Київ", 92.5],  
    ["Іван", 20, "Львів", 85.3],  
    ["Марія", 18, "Харків", 94.8],  
    ["Петро", 21, "Одеса", 78.9]  
]
```

```
# Запис у CSV
```

```
with open("students.csv", "w", newline="", encoding="utf-8") as f:  
    writer = csv.writer(f)  
    writer.writerows(students)  
print("CSV-файл створено")
```

```
# Читання з CSV
```

```
with open("students.csv", "r", encoding="utf-8") as f:  
    reader = csv.reader(f)  
    print("Вміст CSV:")  
    for i, row in enumerate(reader):  
        print(f" Рядок {i}: {row}")
```

```
# Читання як словників
```

```
with open("students.csv", "r", encoding="utf-8") as f:  
    reader = csv.DictReader(f)  
    print("\nЯк словники:")  
    for row in reader:  
        print(f"    {row['Ім'я']}, {row['Вік']} років, {row['Місто']}, бал  
{row['Середній бал']}")
```

Модуль pathlib – сучасна робота з шляхами

pathlib – це сучасний, об’єктно-орієнтований спосіб роботи з шляхами.

```
from pathlib import Path
```

```
print("\n■ МОДУЛЬ PATHLIB")
```

```
print("=" * 60)
```

```
# Створення об’єкта Path
```

```
current = Path(".")
```

```
print(f"Поточний шлях: {current.absolute()}")
```

```
# Робота з шляхами
```

```
home = Path.home()
```

```
print(f"Домашня директорія: {home}")
```

```
# Конкатенація шляхів
```

```
docs = home / "Documents" / "project" / "file.txt"
```

```
print(f"Шлях до файлу: {docs}")
```

```
# Властивості шляху
```

```
print(f"Батьківська директорія: {docs.parent}")
```

```
print(f"Ім’я файлу: {docs.name}")
```

```
print(f"Стем (без розширення): {docs.stem}")
```

```
print(f"Розширення: {docs.suffix}")
```

```
# Перевірка існування
```

```
test_file = Path("sample.txt")
```

```
print(f"\nФайл sample.txt існує: {test_file.exists()}")
```

```
print(f"Це файл? {test_file.is_file()}")
```

```
print(f"Це директорія? {test_file.is_dir()}")
```

```
# Читання/запис за допомогою pathlib
```

```
data = "Текст для запису через pathlib"
```

```
test_file.write_text(data, encoding="utf-8")
```

```
print(f"\nЗаписано текст у {test_file}")
```

```
print(f"Прочитано: {test_file.read_text(encoding='utf-8')}")
```

Створення директорій

```
new_dir = Path("test_dir/sub_dir")
new_dir.mkdir(parents=True, exist_ok=True)
print(f"\nСтворено директорію: {new_dir}")
```

Пошук файлів

```
print("\nВсі .txt файли в поточній директорії:")
for txt_file in Path(".").glob("*.txt"):
    print(f" {txt_file}")
```

Видалення (обережно!)

new_dir.rmdir() # видаляє тільки порожню директорію

Обробка винятків при роботі з файлами

```
print("\n⚠ ОБРОБКА ПОМИЛОК")
print("=" * 60)
```

```
def safe_read_file(filename):
    """Безпечне читання файлу з обробкою помилок"""
    try:
        with open(filename, "r", encoding="utf-8") as f:
            return f.read()
    except FileNotFoundError:
        print(f"✗ Файл {filename} не знайдено")
        return None
    except PermissionError:
        print(f"✗ Немає прав на читання файлу {filename}")
        return None
    except UnicodeDecodeError:
        print(f"✗ Помилка кодування при читанні {filename}")
        return None
    except Exception as e:
        print(f"✗ Невідома помилка: {e}")
        return None
```

Тестуємо

```
content = safe_read_file("неіснуючий_файл.txt")
print(content)
```

Практичний приклад: програма для ведення нотаток

```
import json
import os
from datetime import datetime

class NoteManager:
    """Менеджер нотаток із збереженням у файл"""

    def __init__(self, filename="notes.json"):
        self.filename = filename
        self.notes = self.load_notes()

    def load_notes(self):
        """Завантажує нотатки з файлу"""
        if not os.path.exists(self.filename):
            return []

        try:
            with open(self.filename, "r", encoding="utf-8") as f:
                return json.load(f)
        except:
            return []

    def save_notes(self):
        """Зберігає нотатки у файл"""
        with open(self.filename, "w", encoding="utf-8") as f:
            json.dump(self.notes, f, ensure_ascii=False, indent=4)

    def add_note(self, title, content):
        """Додає нову нотатку"""
        note = {
            "id": len(self.notes) + 1,
            "title": title,
            "content": content,
```

```

        "created": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        "updated": datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    }
    self.notes.append(note)
    self.save_notes()
    print(f"✓ Нотатку '{title}' додано!")

def list_notes(self):
    """Показує всі нотатки"""
    if not self.notes:
        print("📁 Нотаток немає")
        return

    print("\n📁 ВАШІ НОТАТКИ:")
    for note in self.notes:
        print(f"{note['id']}. {note['title']} ({note['created']})")

def show_note(self, note_id):
    """Показує конкретну нотатку"""
    for note in self.notes:
        if note['id'] == note_id:
            print(f"\n📝 {note['title']}")
            print(f"📅 Створено: {note['created']}")
            print(f"📅 Оновлено: {note['updated']}")
            print("-" * 40)
            print(note['content'])
            return
    print(f"✗ Нотатку з ID {note_id} не знайдено")

def delete_note(self, note_id):
    """Видаляє нотатку"""
    for i, note in enumerate(self.notes):
        if note['id'] == note_id:
            deleted = self.notes.pop(i)
            self.save_notes()
            print(f"✓ Нотатку '{deleted['title']}' видалено")

```

```

        return

    print(f"✗ Нотатку з ID {note_id} не знайдено")

def main():
    """Головна функція програми"""
    manager = NoteManager()

    print("📌 МЕНЕДЖЕР НОТАТОК")
    print("=" * 50)

    while True:
        print("\nКоманди:")
        print(" 1. Додати нотатку")
        print(" 2. Показати всі нотатки")
        print(" 3. Показати нотатку")
        print(" 4. Видалити нотатку")
        print(" 5. Вийти")

        choice = input("Оберіть команду (1-5): ").strip()

        if choice == "1":
            title = input("Заголовок: ").strip()
            print("Текст (порожній рядок для завершення):")
            lines = []
            while True:
                line = input()
                if not line:
                    break
                lines.append(line)
            content = "\n".join(lines)
            manager.add_note(title, content)

        elif choice == "2":
            manager.list_notes()

        elif choice == "3":
            try:

```



```

        note_id = int(input("ID нотатки: "))
        manager.show_note(note_id)
    except ValueError:
        print("✗ Введіть число")

elif choice == "4":
    try:
        note_id = int(input("ID нотатки для видалення: "))
        manager.delete_note(note_id)
    except ValueError:
        print("✗ Введіть число")

elif choice == "5":
    print("👋 До побачення!")
    break

else:
    print("✗ Невідома команда")

if __name__ == "__main__":
    main()

```

Цей приклад демонструє:

- Збереження даних у JSON-файл між запусками
- Використання with для безпечної роботи з файлами
- Структурування коду в класи
- Обробку помилок
- Інтерактивний інтерфейс командного рядка

У наступній лекції ми перейдемо до структур даних та алгоритмів – фундаментальних концепцій, які лежать в основі ефективного програмування.

Питання для обговорення

1. Поясніть різницю між модулем та пакетом у Python. Як створити власний модуль? Як створити пакет і для чого потрібен файл `__init__.py`?
2. Які є способи імпорту модулів в Python? Поясніть різницю між `import math`, `from math import sqrt` та `from math import *`. Чому використання `import *` вважається поганою практикою?

3. Проаналізуйте код:

```
import math as m
print(m.sqrt(16))
```

Для чого використовується ключове слово `as`? У яких ситуаціях це може бути корисно?

4. Що таке змінна `__name__`? Поясніть конструкцію `if __name__ == "__main__":`. Чому вона важлива при створенні модулів, які можна як імпортувати, так і запускати як окрему програму?

5. Напишіть невеликий модуль з функціями для роботи з рядками (наприклад, `reverse_string`, `count_vowels`). Потім напишіть іншу програму, яка імпортує цей модуль і використовує його функції.

6. Як модуль `random` генерує випадкові числа? Чому важливо іноді використовувати `random.seed()`? Наведіть приклад, де це може бути корисним.

7. Поясніть різницю між режимами відкриття файлів `'r'`, `'w'`, `'a'`, `'r+'`. Що станеться, якщо відкрити неіснуючий файл у режимі `'r'`? А в режимі `'w'`?

8. Чому конструкція `with open(...) as f:` вважається кращою за звичайне відкриття файлу без неї? Що гарантує контекстний менеджер навіть у разі виникнення помилки?

9. Як прочитати з файлу всі рядки в список? Як записати список рядків у файл? Наведіть приклади з використанням методів `readlines()` та `writelines()`.

10. Як зберегти структуру даних (наприклад, словник) у файл, щоб потім її можна було відновити? Які формати для цього підходять? Наведіть приклад збереження та відновлення даних за допомогою модуля `json`.

ЛЕКЦІЯ 12

Базові структури даних та алгоритми: стек і черга, сортування, пошук

План лекції

1. Стек (Stack): принцип LIFO та його реалізація.

Почнемо з фундаментальної структури даних – стеку. Пояснимо принцип LIFO (Last In, First Out) на життєвих прикладах. Розберемо основні операції: push, pop, peek. Покажемо, як реалізувати стек на основі списку Python і де він застосовується: перевірка дужок, історія браузера, скасування дій.

2. Черга (Queue): принцип FIFO та різновиди.

Розглянемо чергу – структуру з принципом FIFO (First In, First Out). Покажемо реалізацію на списку та її недоліки. Познайомимо з ефективною реалізацією через collections.deque. Розберемо двосторонню чергу (deque) та пріоритетну чергу.

3. Алгоритми сортування: бульбашка, вибір, вставки.

Перейдемо до алгоритмів – покрокових інструкцій для розв'язання задач. Розберемо три прості алгоритми сортування: бульбашкою, вибором та вставками. Пояснимо їх принципи, реалізуємо, проаналізуємо ефективність. Порівняємо з вбудованим сортуванням Python.

4. Алгоритми пошуку: лінійний та бінарний.

Навчимося шукати елементи в масивах. Розберемо лінійний пошук – простий, але неефективний для великих даних. Потім бінарний пошук – набагато швидший, але вимагає відсортованих даних. Реалізуємо обидва, включаючи рекурсивну версію бінарного пошуку.

1. Стек (Stack): принцип LIFO та його реалізація

Стек – це одна з найфундаментальніших структур даних у програмуванні. Назва походить від аналогії зі стосом тарілок: ви кладете тарілки одна на одну, і коли вам потрібна тарілка, ви берете верхню – ту, яку поклали останньою. Саме тому принцип роботи стеку називають **LIFO – Last In, First Out** (останнім прийшов – першим вийшов).

Життєві аналогії стеку

- Стопка книг: ви кладете книгу зверху і берете теж зверху
- Історія браузера: кнопка "Назад" повертає вас на останню відвідану сторінку
- Скасування дій (Ctrl+Z): остання дія скасовується першою
- Залізничний тупик: вагони заїжджають і виїжджають в одному порядку

Основні операції зі стеком

Будь-який стек підтримує три основні операції:

- **Push** – додати елемент на вершину стеку
- **Pop** – видалити елемент з вершини і повернути його
- **Peek** (або Top) – подивитися на верхній елемент без видалення

Реалізація стеку на основі списку Python

У Python список чудово підходить для реалізації стеку. Ми будемо вважати, що кінець списку – це вершина стеку.

```
class Stack:
    """Реалізація стеку на основі списку"""

    def __init__(self):
        self.items = []

    def push(self, item):
        """Додає елемент на вершину стеку"""
        self.items.append(item)

    def pop(self):
        """Видаляє та повертає елемент з вершини стеку"""
        if self.is_empty():
            raise IndexError("Стек порожній")
        return self.items.pop()

    def peek(self):
        """Повертає верхній елемент без видалення"""
        if self.is_empty():
            raise IndexError("Стек порожній")
        return self.items[-1]

    def is_empty(self):
        """Перевіряє, чи стек порожній"""
        return len(self.items) == 0

    def size(self):
        """Повертає кількість елементів у стеку"""
        return len(self.items)
```

```

def __str__(self):
    return f"Stack: {self.items}"

# Демонстрація роботи стеку
stack = Stack()
print("Створено порожній стек")

# Додаємо елементи
stack.push(10)
stack.push(20)
stack.push(30)
print(f"Після додавання: {stack}")
print(f"Розмір стеку: {stack.size()}")
print(f"Верхній елемент: {stack.peek()}")

# Видаляємо елементи
print(f"Видалено: {stack.pop()}")
print(f"Видалено: {stack.pop()}")
print(f"Після видалення: {stack}")
print(f"Стек порожній? {stack.is_empty()}")

```

Практичне застосування 1: Перевірка симетричності дужок

Одне з класичних застосувань стеку – перевірка правильності вкладеності дужок у тексті.

```

def check_brackets(text):
    """
    Перевіряє, чи правильно розставлені дужки в тексті.
    Підтримує (), [], {}
    """
    stack = Stack()
    brackets = {'(': ')', '[': ']', '{': ' '}
    opening = brackets.values()
    closing = brackets.keys()

    for i, char in enumerate(text):
        if char in opening:

```

```

        stack.push((char, i)) # зберігаємо символ і позицію
    elif char in closing:
        if stack.is_empty():
            return False, f"Закриваюча дужка '{char}' на позиції {i} без
відкриваючої"

        last_open, pos = stack.pop()
        if last_open != brackets[char]:
            return False, f"Невідповідність дужок: '{last_open}' на позиції
{pos} та '{char}' на позиції {i}"

        if not stack.is_empty():
            last_open, pos = stack.pop()
            return False, f"Незакрита відкриваюча дужка '{last_open}' на позиції
{pos}"

    return True, "Дужки розставлені правильно"

```

Тестуємо

```

test_strings = [
    "(a + b) * (c - d)",
    "{[()]} ",
    "({[]})",
    "(( ))",
    "function() { return [1, 2, 3]; }"
]

for s in test_strings:
    is_valid, message = check_brackets(s)
    print(f"'{s[:30]}'... -> {message}")

```

Практичне застосування 2: Історія браузера

```

class BrowserHistory:
    """Імітація історії браузера з можливістю повернення назад"""

    def __init__(self):
        self.back_stack = Stack() # для зберігання історії назад

```

```
self.forward_stack = Stack() # для зберігання історії вперед  
self.current_page = None
```

```
def visit(self, url):  
    """Відвідує нову сторінку"""  
    if self.current_page:  
        self.back_stack.push(self.current_page)  
        # Очищуємо forward stack, бо це нова гілка історії  
        self.forward_stack = Stack()  
  
    self.current_page = url  
    print(f"📍 Відвідуємо: {url}")
```

```
def back(self):  
    """Повертається на попередню сторінку"""  
    if self.back_stack.is_empty():  
        print("❌ Немає історії назад")  
        return None  
  
    self.forward_stack.push(self.current_page)  
    self.current_page = self.back_stack.pop()  
    print(f"← Назад: {self.current_page}")  
    return self.current_page
```

```
def forward(self):  
    """Переходить вперед по історії"""  
    if self.forward_stack.is_empty():  
        print("❌ Немає історії вперед")  
        return None  
  
    self.back_stack.push(self.current_page)  
    self.current_page = self.forward_stack.pop()  
    print(f"→ Вперед: {self.current_page}")  
    return self.current_page
```

```
def show_history(self):  
    """Показує поточний стан"""
```

```

print(f"\n🏠 Історія назад: {self.back_stack}")
print(f"👉 Поточна сторінка: {self.current_page}")
print(f"🏠 Історія вперед: {self.forward_stack}")

```

Демонстрація

```

browser = BrowserHistory()
browser.visit("google.com")
browser.visit("youtube.com")
browser.visit("github.com")
browser.back()
browser.back()
browser.forward()
browser.show_history()

```

Практичне застосування 3: Обчислення виразу в зворотному польському записі

Зворотний польський запис (RPN) – це спосіб запису виразів, де оператори йдуть після операндів. Наприклад, "3 4 +" означає $3 + 4$. Стек ідеально підходить для обчислення таких виразів.

```

def evaluate_rpn(expression):
    """
    Обчислює вираз у зворотному польському записі.
    Наприклад: "3 4 + 2 *" -> (3+4)*2 = 14
    """
    stack = Stack()
    operators = {'+', '-', '*', '/', '^'}

    for token in expression.split():
        if token not in operators:
            # Це число, додаємо в стек
            stack.push(float(token))
        else:
            # Це оператор, беремо два останні числа
            if stack.size() < 2:
                raise ValueError("Недостатньо операндів")

```



```

b = stack.pop()
a = stack.pop()

if token == '+':
    result = a + b
elif token == '-':
    result = a - b
elif token == '*':
    result = a * b
elif token == '/':
    if b == 0:
        raise ValueError("Ділення на нуль")
    result = a / b
elif token == '^':
    result = a ** b

stack.push(result)

if stack.size() != 1:
    raise ValueError("Помилка у виразі")

return stack.pop()

# Тестуємо
expressions = [
    "3 4 +",      # 7
    "5 2 -",      # 3
    "10 3 * 4 +",  # 34
    "3 4 + 2 *",   # 14
    "15 7 1 1 + - / 3 * 2 1 1 + + -" # складний вираз
]

for expr in expressions:
    try:
        result = evaluate_rpn(expr)
        print(f"'{expr}' = {result}")
    except Exception as e:

```

```
print(f'Помилка в '{expr}': {e}')
```

2. Черга (Queue): принцип FIFO та різновиди

Черга – це структура даних, яка працює за принципом **FIFO** – **First In, First Out** (першим прийшов – першим вийшов). Це точно як звичайна черга в магазині, банку або їдальні: хто прийшов перший, той обслуговується першим.

Життєві аналогії черги

- Черга в магазині
- Завдання на принтері (друкує в порядку надходження)
- Обробка повідомлень у месенджері
- Автомобілі на мийці

Основні операції з чергою

- **Enqueue** – додати елемент у кінець черги
- **Dequeue** – видалити елемент з початку черги і повернути його
- **Front** (або **Peek**) – подивитися на перший елемент без видалення

Реалізація черги на основі списку (неефективна)

Спочатку спробуємо реалізувати чергу звичайним списком, але побачимо проблему:

```
class SimpleQueue:
    """Проста, але неефективна реалізація черги"""

    def __init__(self):
        self.items = []

    def enqueue(self, item):
        """Додає елемент у кінець черги"""
        self.items.append(item)

    def dequeue(self):
        """Видаляє та повертає елемент з початку черги"""
        if self.is_empty():
            raise IndexError("Черга порожня")
        return self.items.pop(0) # Проблема: O(n) – повільно!
```

```

def front(self):
    """Повертає перший елемент без видалення"""
    if self.is_empty():
        raise IndexError("Черга порожня")
    return self.items[0]

def is_empty(self):
    return len(self.items) == 0

def size(self):
    return len(self.items)

def __str__(self):
    return f"Queue: {self.items}"

```

Демонстрація

```

queue = SimpleQueue()
queue.enqueue("Клієнт 1")
queue.enqueue("Клієнт 2")
queue.enqueue("Клієнт 3")
print(f"Черга: {queue}")
print(f"Обслуговується: {queue.dequeue()}")
print(f"Черга після обслуговування: {queue}")
print(f"Наступний: {queue.front()}")

```

Чому `pop(0)` неефективний? Тому що після видалення першого елемента всі інші елементи мають зміститися вліво. Це операція $O(n)$ – лінійна складність. Для великих черг це буде дуже повільно.

Ефективна реалізація за допомогою `collections.deque`

Модуль `collections` надає структуру `deque` (double-ended queue), яка оптимізована для швидкого додавання та видалення з обох кінців.

```

from collections import deque

class EfficientQueue:
    """Ефективна реалізація черги за допомогою deque"""

```

```

def __init__(self):
    self.items = deque()

def enqueue(self, item):
    """Додає елемент у кінець черги – O(1)"""
    self.items.append(item)

def dequeue(self):
    """Видаляє та повертає елемент з початку черги – O(1)"""
    if self.is_empty():
        raise IndexError("Черга порожня")
    return self.items.popleft()

def front(self):
    """Повертає перший елемент без видалення – O(1)"""
    if self.is_empty():
        raise IndexError("Черга порожня")
    return self.items[0]

def is_empty(self):
    return len(self.items) == 0

def size(self):
    return len(self.items)

def __str__(self):
    return f"Queue: {list(self.items)}"

# Демонстрація швидкості
import time

def test_queue_performance(queue_class, n=10000):
    """Тестує швидкість роботи черги"""
    q = queue_class()

    # Тест додавання
    start = time.time()

```

```

for i in range(n):
    q.enqueue(i)
enqueue_time = time.time() - start

# Тест видалення
start = time.time()
for i in range(n):
    q.dequeue()
dequeue_time = time.time() - start

return enqueue_time, dequeue_time

# Порівнюємо SimpleQueue і EfficientQueue
print("Порівняння продуктивності (n=10000):")
simple_time = test_queue_performance(SimpleQueue)
print(f"SimpleQueue - додавання: {simple_time[0]:.4f}с, видалення: {simple_time[1]:.4f}с")

efficient_time = test_queue_performance(EfficientQueue)
print(f"EfficientQueue - додавання: {efficient_time[0]:.4f}с, видалення: {efficient_time[1]:.4f}с")

```

Двостороння черга (Deque)

Deque дозволяє ефективно додавати та видаляти елементи з обох кінців:

```

from collections import deque

print("🌀 ДВОСТОРОННЯ ЧЕРГА (DEQUE)")
print("=" * 60)

# Створення deque
dq = deque([1, 2, 3, 4, 5])
print(f"Початковий deque: {dq}")

# Додавання з різних кінців
dq.append(6)      # додати праворуч
dq.appendleft(0)  # додати ліворуч
print(f"Після додавання: {dq}")

```

Видалення з різних кінців

```
right = dq.pop()    # видалити праворуч
```

```
left = dq.popleft() # видалити ліворуч
```

```
print(f"Видалено: {left} (зліва), {right} (справа)")
```

```
print(f"Після видалення: {dq}")
```

Розширення

```
dq.extend([6, 7, 8])    # додати список праворуч
```

```
dq.extendleft([-2, -1, 0]) # додати список ліворуч (в зворотному порядку!)
```

```
print(f"Після extend: {dq}")
```

Обертання

```
dq.rotate(2)    # зсунути праворуч на 2
```

```
print(f"Після rotate(2): {dq}")
```

```
dq.rotate(-3)    # зсунути ліворуч на 3
```

```
print(f"Після rotate(-3): {dq}")
```

Практичне застосування 1: Симуляція черги обслуговування

```
import random
```

```
import time
```

```
class ServiceQueue:
```

```
    """Симуляція черги обслуговування клієнтів"""
```

```
    def __init__(self):
```

```
        self.queue = EfficientQueue()
```

```
        self.served_count = 0
```

```
        self.total_wait_time = 0
```

```
    def add_customer(self, name):
```

```
        """Додає клієнта в чергу"""
```

```
        customer = {
```

```
            'name': name,
```

```
            'arrival_time': time.time(),
```

```
            'service_time': random.uniform(1, 3) # час обслуговування 1-3 сек
```

```
        }
```

```

self.queue.enqueue(customer)
print(f"➕ {name} приєднався до черги")

def serve_next(self):
    """Обслуговує наступного клієнта"""
    if self.queue.is_empty():
        print("✖ Черга порожня")
        return None

    customer = self.queue.dequeue()
    wait_time = time.time() - customer['arrival_time']
    self.served_count += 1
    self.total_wait_time += wait_time

    print(f"💰 Обслуговується: {customer['name']}")
    print(f"□ Час очікування: {wait_time:.2f} сек")

    # Симулюємо обслуговування
    time.sleep(customer['service_time'])

    print(f"✔ {customer['name']} обслужений за {customer['service_time']:.2f} сек")
    return customer

def get_stats(self):
    """Повертає статистику"""
    if self.served_count == 0:
        return "Немає даних"

    avg_wait = self.total_wait_time / self.served_count
    return {
        'served': self.served_count,
        'in_queue': self.queue.size(),
        'avg_wait': avg_wait
    }

# Симуляція

```

```
print("🏠 СИМУЛЯЦІЯ ЧЕРГИ ОБСЛУГОВУВАННЯ")
print("=" * 60)
```

```
service = ServiceQueue()
```

```
# Додаємо клієнтів
```

```
service.add_customer("Олена")
```

```
time.sleep(0.5)
```

```
service.add_customer("Іван")
```

```
time.sleep(0.3)
```

```
service.add_customer("Марія")
```

```
time.sleep(0.2)
```

```
service.add_customer("Петро")
```

```
print("\n📊 Початкова статистика:", service.get_stats())
```

```
print("\nПочинаємо обслуговування...\n")
```

```
# Обслуговуємо всіх
```

```
while not service.queue.is_empty():
```

```
    service.serve_next()
```

```
    print("-" * 40)
```

```
print("\n📊 Фінальна статистика:", service.get_stats())
```

Практичне застосування 2: Пріоритетна черга

Іноді потрібна черга, де елементи обробляються не за часом надходження, а за пріоритетом. Наприклад, у лікарні пацієнти з тяжкими травмами обслуговуються першими.

```
import heapq
```

```
class PriorityQueue:
```

```
    """
```

```
    Пріоритетна черга на основі heapq.
```

```
    Елементи з меншим пріоритетом (числом) обробляються першими.
```

```
    """
```



```

def __init__(self):
    self.heap = []
    self.count = 0

def push(self, item, priority):
    """Додає елемент із заданим пріоритетом"""
    # heapq створює мін-кupu, тому менший пріоритет = вищий
    пріоритет
    heapq.heappush(self.heap, (priority, self.count, item))
    self.count += 1

def pop(self):
    """Видаляє та повертає елемент з найвищим пріоритетом"""
    if self.is_empty():
        raise IndexError("Черга порожня")
    priority, count, item = heapq.heappop(self.heap)
    return item

def peek(self):
    """Повертає елемент з найвищим пріоритетом без видалення"""
    if self.is_empty():
        raise IndexError("Черга порожня")
    return self.heap[0][2]

def is_empty(self):
    return len(self.heap) == 0

def size(self):
    return len(self.heap)

# Демонстрація
print("🏥 ПРІОРИТЕТНА ЧЕРГА (ЛІКАРНЯ)")
print("=" * 60)

hospital = PriorityQueue()

# Додаємо пацієнтів (менше число = вищий пріоритет)

```

```

hospital.push("Іваненко (перелом)", priority=2)
hospital.push("Петренко (головний біль)", priority=4)
hospital.push("Сидоренко (інфаркт)", priority=1)
hospital.push("Коваленко (застуда)", priority=5)
hospital.push("Бондаренко (кровотеча)", priority=1)

print("Черга пацієнтів за пріоритетом:")
while not hospital.is_empty():
    patient = hospital.pop()
    print(f"➡ {patient}")

```

Практичне застосування 3: Обробка завдань на принтері

```

class PrintQueue:
    """Черга завдань на принтер"""

    def __init__(self):
        self.regular_queue = EfficientQueue() # звичайні завдання
        self.urgent_queue = EfficientQueue() # термінові завдання

    def add_job(self, document, urgent=False):
        """Додає завдання на друк"""
        job = {
            'document': document,
            'time': time.strftime("%H:%M:%S"),
            'pages': random.randint(1, 10)
        }

        if urgent:
            self.urgent_queue.enqueue(job)
            print(f"⚠ Термінове завдання додано: {document}")
        else:
            self.regular_queue.enqueue(job)
            print(f"📄 Звичайне завдання додано: {document}")

    def process_next(self):
        """Обробляє наступне завдання (спочатку термінові)"""

```

```

if not self.urgent_queue.is_empty():
    job = self.urgent_queue.dequeue()
    print(f"\n🔪 Друк термінового: {job['document']}")
elif not self.regular_queue.is_empty():
    job = self.regular_queue.dequeue()
    print(f"\n📄 Друк звичайного: {job['document']}")
else:
    print("❌ Немає завдань")
    return None

```

```

print(f" ⌚ Час додавання: {job['time']}")
print(f" 📄 Кількість сторінок: {job['pages']}")
print(f" 🖨️ Друкується...")
time.sleep(0.5) # симуляція друку
print(f" ✅ Завершено")
return job

```

```

def queue_status(self):
    """Показує стан черги"""
    print(f"\n📋 Стан черги:")
    print(f"   Термінових завдань: {self.urgent_queue.size()}")
    print(f"   Звичайних завдань: {self.regular_queue.size()}")

```

Демонстрація

```

print("🖨️ СИСТЕМА КЕРУВАННЯ ДРУКОМ")
print("=" * 60)

```

```

printer = PrintQueue()

```

Додаємо завдання

```

printer.add_job("Звіт за квартал.pdf")
printer.add_job("Резюме.docx", urgent=True)
printer.add_job("Запрошення.pdf")
printer.add_job("Договір.docx", urgent=True)
printer.add_job("Презентація.pptx")

```

```
printer.queue_status()
```

```
# Обробляємо всі завдання
```

```
print("\n🔧 Починаємо обробку...")
```

```
while not (printer.regular_queue.is_empty() and
printer.urgent_queue.is_empty()):
    printer.process_next()
    time.sleep(0.3)
```

```
printer.queue_status()
```

У наступних пунктах ми розглянемо алгоритми сортування та пошуку – фундаментальні інструменти, які ви будете використовувати постійно.

3. Алгоритми сортування: бульбашка, вибір, вставки

Сортування – це процес упорядкування елементів у певному порядку (зростання, спадання, за якоюсь ознакою). Існує багато алгоритмів сортування, кожен з яких має свої переваги та недоліки. Ми розглянемо три прості алгоритми, які чудово підходять для навчання: сортування бульбашкою, сортування вибором та сортування вставками.

Чому важливо знати алгоритми сортування?

Хоча в Python є вбудовані функції `sort()` та `sorted()`, які працюють дуже ефективно, розуміння алгоритмів сортування важливе з кількох причин:

- Розвиває алгоритмічне мислення
- Допомагає зрозуміти поняття складності алгоритмів (Big O)
- Дає базу для вивчення складніших алгоритмів
- Допомагає приймати правильні рішення при виборі структур даних

Підготовка даних для тестування

Створимо допоміжні функції для тестування наших алгоритмів:

```
import random
```

```
import time
```

```
def generate_random_list(size, min_val=0, max_val=100):
    """Генерує випадковий список заданого розміру"""
```

```

return [random.randint(min_val, max_val) for _ in range(size)]

def is_sorted(arr):
    """Перевіряє, чи відсортований список за зростанням"""
    return all(arr[i] <= arr[i+1] for i in range(len(arr)-1))

def test_sorting_algorithm(sort_func, arr):
    """Тестує алгоритм сортування та вимірює час"""
    arr_copy = arr.copy()
    start = time.time()
    sort_func(arr_copy)
    end = time.time()

    if is_sorted(arr_copy):
        print(f"✓ {sort_func.__name__}: {len(arr)} ел., час: {(end-
start)*1000:.2f} мс")
    else:
        print(f"✗ {sort_func.__name__}: ПОМИЛКА – масив не відсортовано")

    return arr_copy

# Створимо тестові дані
test_data = generate_random_list(20, 1, 100)
print(f"Оригінальний масив: {test_data}\n")

```

1. Сортування бульбашкою (Bubble Sort)

Це найпростіший для розуміння алгоритм сортування. Він багаторазово проходить по масиву, порівнюючи сусідні елементи і міняючи їх місцями, якщо вони стоять у неправильному порядку. Великі елементи "спливають" у кінець масиву, наче бульбашки.

```

def bubble_sort(arr):
    """
    Сортування бульбашкою.
    Складність: O(n²) в гіршому випадку, O(n) в кращому (якщо масив уже
    відсортований)
    """

```

```

n = len(arr)

for i in range(n - 1):
    # Прапор для оптимізації: якщо не було обмінів, масив уже
    відсортовано
    swapped = False

    for j in range(n - 1 - i): # -i, бо останні i елементів уже на місці
        if arr[j] > arr[j + 1]:
            # Міняємо місцями
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            swapped = True

    # Якщо не було обмінів, виходимо достроково
    if not swapped:
        break

return arr

# Демонстрація роботи
def bubble_sort_with_trace(arr):
    """Версія з виведенням проміжних результатів"""
    arr = arr.copy()
    n = len(arr)
    print(f'Початковий: {arr}')

    for i in range(n - 1):
        swapped = False
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        print(f'Після проходу {i+1}: {arr}')
        if not swapped:
            print(f'Масив відсортовано достроково!')
            break

```

```
return arr
```

```
print("● СОРТУВАННЯ БУЛЬБАШКОЮ")  
print("=" * 60)  
bubble_sort_with_trace([64, 34, 25, 12, 22, 11, 90])
```

2. Сортвання вибором (Selection Sort)

Алгоритм знаходить мінімальний елемент у невідсортованій частині масиву і міняє його місцями з першим елементом цієї частини. Потім зсуває межу відсортованої частини.

```
def selection_sort(arr):  
    """  
    Сортвання вибором.  
    Складність:  $O(n^2)$  завжди, незалежно від вхідних даних  
    """  
    n = len(arr)  
  
    for i in range(n - 1):  
        # Знаходимо індекс мінімального елемента в невідсортованій частині  
        min_index = i  
        for j in range(i + 1, n):  
            if arr[j] < arr[min_index]:  
                min_index = j  
  
        # Міняємо місцями поточний і мінімальний елементи  
        if min_index != i:  
            arr[i], arr[min_index] = arr[min_index], arr[i]  
  
    return arr  
  
def selection_sort_with_trace(arr):  
    """Версія з виведенням проміжних результатів"""  
    arr = arr.copy()  
    n = len(arr)  
    print(f"Початковий: {arr}")  
  
    for i in range(n - 1):
```

```

min_index = i
for j in range(i + 1, n):
    if arr[j] < arr[min_index]:
        min_index = j

if min_index != i:
    arr[i], arr[min_index] = arr[min_index], arr[i]
    print(f"Міняємо {arr[min_index]} (поз.{min_index}) з {arr[i]} (поз.{i}): {arr}")
else:
    print(f"Елемент {arr[i]} вже на своєму місці")

return arr

print("\n□ СОРТУВАННЯ ВИБОРОМ")
print("=" * 60)
selection_sort_with_trace([64, 34, 25, 12, 22, 11, 90])

```

3. Сорткування вставками (Insertion Sort)

Алгоритм будує відсортований масив поступово, беручи елементи з невідсортованої частини і вставляючи їх у правильну позицію у відсортованій частині. Це дуже схоже на те, як люди сортують карти в руці.

```

def insertion_sort(arr):
    """
    Сорткування вставками.
    Складність:  $O(n^2)$  в гіршому випадку,  $O(n)$  в кращому (для майже
    відсортованих)
    """
    n = len(arr)

    for i in range(1, n):
        key = arr[i] # елемент, який вставляємо
        j = i - 1

        # Зсуваємо елементи вправо, поки не знайдемо місце для key
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]

```



```

    j -= 1

    # Вставляємо ключ на правильне місце
    arr[j + 1] = key

    return arr

def insertion_sort_with_trace(arr):
    """Версія з виведенням проміжних результатів"""
    arr = arr.copy()
    n = len(arr)
    print(f'Початковий: {arr}')

    for i in range(1, n):
        key = arr[i]
        j = i - 1

        print(f'Вставляємо {key} з позиції {i}', end=" ")

        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key
        print(f'→ в позицію {j+1}: {arr}')

    return arr

print("\n● СОРТУВАННЯ ВСТАВКАМИ")
print("=" * 60)
insertion_sort_with_trace([64, 34, 25, 12, 22, 11, 90])

```

Порівняння алгоритмів сортування

Тепер порівняємо всі три алгоритми на різних наборах даних:

```

def compare_sorting_algorithms():
    """Порівнює продуктивність різних алгоритмів сортування"""

```

```

sizes = [100, 500, 1000, 2000]
algorithms = [
    ("Bubble Sort", bubble_sort),
    ("Selection Sort", selection_sort),
    ("Insertion Sort", insertion_sort),
    ("Python Timsort", lambda x: x.sort()) # вбудоване сортування
]

print("■ ПОРІВНЯННЯ АЛГОРИТМІВ СОРТУВАННЯ")
print("=" * 80)
print(f'{"Алгоритм":<15} {"Розмір":<10} {"Час (мс)":<15} {"Відсортовано":<10}')
```

```

print("-" * 80)

for size in sizes:
    # Випадковий масив
    test_array = generate_random_list(size)

    for name, algorithm in algorithms:
        arr_copy = test_array.copy()

        start = time.time()
        algorithm(arr_copy)
        end = time.time()

        sorted_correctly = is_sorted(arr_copy)
        time_ms = (end - start) * 1000

        print(f'{name:<15} {size:<10} {time_ms:<15.2f} {✓ if
sorted_correctly else ✗}')
```

Порівнюємо (обережно: bubble sort на 2000 елементів може бути дуже повільним!)

compare_sorting_algorithms()

Аналіз складності алгоритмів

Алгоритм	Кращий випадок	Середній випадок	Гірший випадок	Пам'ять	Стабільність
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Так
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ні
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Так
Python Timsort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Так

Коли який алгоритм використовувати?

- **Bubble Sort:** тільки для навчання, майже ніколи в реальних проектах
- **Selection Sort:** коли важлива мінімальна кількість обмінів (наприклад, для даних, які дорого копіювати)
- **Insertion Sort:** для малих масивів (до 20-30 елементів) або для майже відсортованих даних
- **Timsort (вбудований):** для всього іншого – він оптимізований для реальних даних

4. Алгоритми пошуку: лінійний та бінарний

Пошук елементів у масиві – ще одна фундаментальна задача. Ми розглянемо два основні алгоритми: лінійний пошук (простий, але повільний) та бінарний пошук (швидкий, але вимагає відсортованих даних).

Лінійний пошук (Linear Search)

Найпростіший алгоритм пошуку: проходимо по масиву від початку до кінця і порівнюємо кожен елемент з шуканим.

```
def linear_search(arr, target):
```

```
    """
```

Лінійний пошук.

Повертає індекс першого входження target або -1, якщо не знайдено.

Складність: $O(n)$

```
"""
```

```
for i in range(len(arr)):
```

```
    if arr[i] == target:
```

```
        return i
```

```
return -1
```

```
def linear_search_with_trace(arr, target):
```

```
    """Версія з виведенням процесу пошуку"""
```

```
    print(f"Шукаємо {target} в масиві: {arr}")
```

```
    for i in range(len(arr)):
```

```
        print(f"Перевіряємо індекс {i}: значення {arr[i]}")
```

```
        if arr[i] == target:
```

```
            print(f"✓ Знайдено на позиції {i}")
```

```
            return i
```

```
    print(f"✗ Елемент {target} не знайдено")
```

```
    return -1
```

Демонстрація

```
arr = [5, 3, 8, 1, 9, 2, 7, 4, 6]
```

```
print("🔍 ЛІНІЙНИЙ ПОШУК")
```

```
print("=" * 60)
```

```
linear_search_with_trace(arr, 7)
```

```
print()
```

```
linear_search_with_trace(arr, 10)
```

Бінарний пошук (Binary Search)

Бінарний пошук набагато ефективніший, але працює тільки на **відсортованих** масивах. Алгоритм працює за принципом "розділяй і володарюй": порівнює шуканий елемент із середнім елементом масиву і на основі цього відкидає половину масиву.

```
def binary_search(arr, target):
```

```
    """
```

Ітеративний бінарний пошук.

Повертає індекс target або -1, якщо не знайдено.

Складність: $O(\log n)$

"""

```
left, right = 0, len(arr) - 1
```

```
while left <= right:
```

```
    mid = (left + right) // 2
```

```
    if arr[mid] == target:
```

```
        return mid
```

```
    elif arr[mid] < target:
```

```
        left = mid + 1 # шукаємо в правій половині
```

```
    else:
```

```
        right = mid - 1 # шукаємо в лівій половині
```

```
return -1
```

```
def binary_search_with_trace(arr, target):
```

```
    """Версія з виведенням процесу пошуку"""
```

```
    left, right = 0, len(arr) - 1
```

```
    print(f'Шукаємо {target} у відсортованому масиві: {arr}')
```

```
    step = 1
```

```
    while left <= right:
```

```
        mid = (left + right) // 2
```

```
        print(f"    Крок {step}: left={left}, right={right}, mid={mid},  
arr[{mid}]={arr[mid]}")
```

```
        if arr[mid] == target:
```

```
            print(f"✓ Знайдено на позиції {mid}')
```

```
            return mid
```

```
        elif arr[mid] < target:
```

```
            print(f"    {target} > {arr[mid]}, шукаємо в правій половині')
```

```
            left = mid + 1
```

```
        else:
```

```
            print(f"    {target} < {arr[mid]}, шукаємо в лівій половині')
```

```

    right = mid - 1

    step += 1

    print(f"✗ Елемент {target} не знайдено")
    return -1

# Демонстрація (масив має бути відсортований!)
sorted_arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
print("\n🔍 БІНАРНИЙ ПОШУК")
print("=" * 60)
binary_search_with_trace(sorted_arr, 7)
print()
binary_search_with_trace(sorted_arr, 16)

```

Рекурсивний бінарний пошук

```

def binary_search_recursive(arr, target, left=None, right=None):
    """
    Рекурсивна версія бінарного пошуку.
    """
    if left is None:
        left, right = 0, len(arr) - 1

    if left > right:
        return -1 # базовий випадок: елемент не знайдено

    mid = (left + right) // 2

    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, target, mid + 1, right)
    else:
        return binary_search_recursive(arr, target, left, mid - 1)

print("\n🔄 РЕКУРСИВНИЙ БІНАРНИЙ ПОШУК")

```

```

print("=" * 60)
index = binary_search_recursive(sorted_arr, 7)
print(f"Пошук 7: знайдено на позиції {index}")
index = binary_search_recursive(sorted_arr, 20)
print(f"Пошук 20: результат {index}")

```

Порівняння лінійного та бінарного пошуку

```

def compare_search_algorithms():
    """Порівнює продуктивність лінійного та бінарного пошуку"""

    sizes = [1000, 10000, 100000, 1000000]

    print("\nПОРІВНЯННЯ АЛГОРИТМІВ ПОШУКУ")
    print("=" * 80)
    print(f"{'Алгоритм':<15}      {'Розмір':<10}      {'Час      (мс)':<15}"
          f"{'Результат':<10}")
    print("-" * 80)

    for size in sizes:
        # Створюємо відсортований масив
        arr = list(range(size))

        # Шукаємо елемент, який є (останній)
        target = size - 1

        # Лінійний пошук
        start = time.time()
        linear_result = linear_search(arr, target)
        linear_time = (time.time() - start) * 1000

        # Бінарний пошук
        start = time.time()
        binary_result = binary_search(arr, target)
        binary_time = (time.time() - start) * 1000

        print(f"{'Linear':<15} {size:<10} {linear_time:<15.4f} {linear_result}")
        print(f"{'Binary':<15} {size:<10} {binary_time:<15.4f} {binary_result}")

```

```
print("-" * 40)
```

```
# Порівнюємо
```

```
compare_search_algorithms()
```

Практичне застосування: пошук у словнику

```
class Dictionary:
```

```
    """Простий словник з бінарним пошуком"""
```

```
    def __init__(self):
```

```
        self.words = []
```

```
        self.definitions = []
```

```
    def add_word(self, word, definition):
```

```
        """Додає слово з визначенням, підтримуючи відсортованість"""
```

```
        # Знаходимо місце для вставки (бінарним пошуком)
```

```
        pos = self._find_insert_position(word)
```

```
        self.words.insert(pos, word)
```

```
        self.definitions.insert(pos, definition)
```

```
        print(f"✓ Додано: {word}")
```

```
    def _find_insert_position(self, word):
```

```
        """Знаходить позицію для вставки нового слова"""
```

```
        left, right = 0, len(self.words) - 1
```

```
        while left <= right:
```

```
            mid = (left + right) // 2
```

```
            if self.words[mid] == word:
```

```
                return mid # слово вже є
```

```
            elif self.words[mid] < word:
```

```
                left = mid + 1
```

```
            else:
```

```
                right = mid - 1
```

```
        return left # позиція для вставки
```



```

def find_word(self, word):
    """Знаходить визначення слова (бінарний пошук)"""
    index = binary_search(self.words, word)
    if index != -1:
        return self.definitions[index]
    return None

def __str__(self):
    result = []
    for i in range(len(self.words)):
        result.append(f"{self.words[i]}: {self.definitions[i]}")
    return "\n".join(result)

# Демонстрація
print("📖 ЕЛЕКТРОННИЙ СЛОВНИК")
print("=" * 60)

dictionary = Dictionary()
dictionary.add_word("banana", "жовтий фрукт")
dictionary.add_word("apple", "червоний або зелений фрукт")
dictionary.add_word("cat", "домашня тварина")
dictionary.add_word("dog", "найкращий друг людини")
dictionary.add_word("elephant", "велика сіра тварина")

print("\n📖 Вміст словника:")
print(dictionary)

print("\n🔍 Пошук:")
word = "cat"
definition = dictionary.find_word(word)
print(f"{word}: {definition}")

word = "zebra"
definition = dictionary.find_word(word)
print(f"{word}: {definition}")

```

Практичне застосування: пошук у великих даних

```
def search_students_by_id(students, target_id):
    """
    Пошук студента за ID (припускаємо, що список відсортований за ID)
    """
    left, right = 0, len(students) - 1

    while left <= right:
        mid = (left + right) // 2

        if students[mid]['id'] == target_id:
            return students[mid]
        elif students[mid]['id'] < target_id:
            left = mid + 1
        else:
            right = mid - 1

    return None
```

Створюємо тестові дані

```
students = [
    {'id': 1001, 'name': 'Олена Петренко', 'grade': 92},
    {'id': 1002, 'name': 'Іван Коваленко', 'grade': 85},
    {'id': 1003, 'name': 'Марія Шевченко', 'grade': 95},
    {'id': 1004, 'name': 'Петро Бондаренко', 'grade': 78},
    {'id': 1005, 'name': 'Анна Кравченко', 'grade': 98},
    # ... ще багато студентів
]
```

```
print("👤🎓 ПОШУК СТУДЕНТА ЗА ID")
print("=" * 60)
```

```
target = 1003
student = search_students_by_id(students, target)
```

```
if student:
```

```
print(f"Знайдено: {student['name']} (ID: {student['id']}), оцінка: {student['grade']}")
else:
    print(f"Студента з ID {target} не знайдено")
```

Підсумки

- **Лінійний пошук** простий і працює на будь-яких даних, але повільний для великих масивів ($O(n)$)
- **Бінарний пошук** дуже швидкий ($O(\log n)$), але вимагає відсортованих даних
 - Для пошуку в невеликих масивах (до 100 елементів) різниця непомітна
 - Для великих даних бінарний пошук у тисячі разів швидший
 - Вбудовані структури даних Python (set, dict) використовують хешування і забезпечують пошук за $O(1)$

У наступній лекції ми розглянемо модуль collections та інші розширені структури даних, які роблять Python ще потужнішим.

1. Поясніть принцип роботи стеку (LIFO). Наведіть приклади з реального життя, де використовується цей принцип. Як реалізувати стек на основі списку Python?

2. Поясніть принцип роботи черги (FIFO). Наведіть приклади з реального життя. Чому реалізація черги на основі списку з використанням `pop(0)` є неефективною? Як це виправити за допомогою `collections.deque`?

3. Напишіть функцію, яка використовує стек для перевірки правильності вкладеності дужок у рядку (наприклад, для `"({[]})"` має повертати `True`, а для `"({[]])"` – `False`).

4. Поясніть алгоритм сортування бульбашкою (bubble sort). Чому він має квадратичну складність $O(n^2)$? Як можна оптимізувати цей алгоритм, щоб він завершувався достроково для майже відсортованих масивів?

5. Порівняйте сортування бульбашкою, вибором та вставками. Який з алгоритмів найкраще працює для майже відсортованих даних? Чому?

6. Чому вбудоване сортування Python (`list.sort()` та `sorted()`) набагато швидше за прості алгоритми, які ми розглянули? Що таке Timsort і які його переваги?

7. Поясніть різницю між лінійним та бінарним пошуком. Які умови необхідні для застосування бінарного пошуку? Яка складність кожного алгоритму?

8. Напишіть рекурсивну версію бінарного пошуку. Чому рекурсивна реалізація може бути менш ефективною за ітеративну?

9. Проаналізуйте код:

```
numbers = [1, 3, 5, 7, 9, 11, 13, 15]
target = 7
left, right = 0, len(numbers) - 1
while left <= right:
    mid = (left + right) // 2
    if numbers[mid] == target:
        print(mid)
        break
    elif numbers[mid] < target:
        left = mid + 1
    else:
        right = mid - 1
```

Поясніть, як працює цей код. Чому умова циклу `left <= right`, а не `left < right`?

10. Уявіть, що вам потрібно часто шукати елементи у великому масиві даних. Яку структуру даних ви оберете: список з бінарним пошуком чи множину? Чому? Які компроміси між швидкістю пошуку та використанням пам'яті?

ЛЕКЦІЯ 13

Використання колекцій Python для роботи з даними

План лекції

1. Огляд модуля collections та namedtuple.

Почнемо зі знайомства з модулем collections. Детально розберемо namedtuple – структуру, яка поєднує легкість кортежів зі зручністю іменованих полів. Це ідеальний вибір для створення простих класів без зайвого коду.

2. defaultdict та Counter: словники з надздібностями.

Розглянемо дві надзвичайно корисні структури: defaultdict, який автоматично створює значення для відсутніх ключів, та Counter – спеціалізований лічильник для підрахунку елементів. Покажемо, як вони спрощують код, який без них був би громіздким.

3. OrderedDict, ChainMap та інші корисні структури.

Познайомимось з OrderedDict – словником, який пам'ятає порядок додавання елементів, та ChainMap для об'єднання кількох словників. Розберемо ситуації, де вони незамінні.

4. Вибір оптимальної структури даних та list comprehensions.

Підсумуємо знання про структури даних і навчимося вибирати правильну структуру для конкретної задачі. Також поглибимо знання про list, dict та set comprehensions – елегантні способи створення колекцій. Наприкінці торкнемося генераторів для роботи з великими даними.

1. Огляд модуля collections та namedtuple

Модуль collections – це одна з "перлин" стандартної бібліотеки Python. Він містить спеціалізовані структури даних, які розширюють можливості вбудованих типів (list, dict, set, tuple) і роблять код більш елегантним та ефективним.

Чому варто використовувати collections?

Перш ніж ми заглибимося в деталі, розглянемо проблему, яку вирішує namedtuple. Уявіть, що вам потрібно зберігати інформацію про точку на площині (координати x, y). У вас є кілька варіантів:

Варіант 1: окремі змінні

x1, y1 = 10, 20

x2, y2 = 30, 40

Проблема: важко групувати пов'язані дані

Варіант 2: список або кортеж

```
point1 = [10, 20]
```

```
point2 = (30, 40)
```

Проблема: point[0] – незрозуміло, що це x чи y

Варіант 3: словник

```
point1 = {'x': 10, 'y': 20}
```

Краще, але займає більше пам'яті і трохи повільніше

Варіант 4: клас

```
class Point:
```

```
    def __init__(self, x, y):
```

```
        self.x = x
```

```
        self.y = y
```

```
point1 = Point(10, 20)
```

Добре, але багато коду для простої структури

namedtuple – ідеальне рішення для простих структур

namedtuple з модуля collections створює клас, який поводить себе як кортеж, але дозволяє звертатися до полів за іменами.

```
from collections import namedtuple
```

Створюємо клас Point

```
Point = namedtuple('Point', ['x', 'y'])
```

Тепер можна створювати об'єкти

```
p1 = Point(10, 20)
```

```
p2 = Point(x=30, y=40) # можна з іменованими аргументами
```

```
print(f"Точка p1: {p1}")
```

```
print(f"x координата p1: {p1.x}")
```

```
print(f"y координата p1: {p1.y}")
```

```
print(f"За індексом: p1[0] = {p1[0]}, p1[1] = {p1[1]}")
```

Розпакування працює як для звичайного кортежу

```
x, y = p1
print(f'Розпакування: x={x}, y={y}')
```

Переваги namedtuple:

- **Легкість:** займає стільки ж пам'яті, скільки звичайний кортеж
- **Читабельність:** доступ до полів за іменами замість індексів
- **Незмінність:** як і кортеж, namedtuple незмінний (immutable)
- **Сумісність:** можна використовувати там, де очікується кортеж

Докладніше про створення namedtuple

```
from collections import namedtuple
```

Різні способи вказати поля

```
Person1 = namedtuple('Person', ['name', 'age', 'city'])
```

```
Person2 = namedtuple('Person', 'name age city') # через пробіл
```

```
Person3 = namedtuple('Person', 'name, age, city') # через кому
```

Можна вказати значення за замовчуванням

```
Person = namedtuple('Person', ['name', 'age', 'city'], defaults=['Невідомо', 0, 'Київ'])
```

Створюємо об'єкти

```
p1 = Person('Олена', 19, 'Львів')
```

```
p2 = Person('Іван', 20) # city буде 'Київ' (за замовчуванням)
```

```
p3 = Person('Марія') # age=0, city='Київ'
```

```
print(f"p1: {p1}")
```

```
print(f"p2: {p2}")
```

```
print(f"p3: {p3}")
```

Можна отримати список полів

```
print(f"Поля: {Person._fields}")
```

```
print(f"Значення за замовчуванням: {Person._field_defaults}")
```

Методи namedtuple

```
from collections import namedtuple
```



```
Car = namedtuple('Car', ['brand', 'model', 'year', 'color'])
```

```
my_car = Car('Toyota', 'Camry', 2020, 'black')  
print(f"Моє авто: {my_car}")
```

_asdict() – перетворює на словник

```
car_dict = my_car._asdict()  
print(f"Як словник: {car_dict}")
```

_replace() – створює новий об'єкт із заміненними полями

```
new_car = my_car._replace(year=2022, color='red')  
print(f"Оригінал: {my_car}")  
print(f"Після заміни: {new_car}")
```

_make() – створює з ітерованого об'єкта

```
data = ['Honda', 'Accord', 2019, 'blue']  
another_car = Car._make(data)  
print(f"Зі списку: {another_car}")
```

Практичний приклад: База даних студентів

```
from collections import namedtuple  
import random
```

Визначаємо структуру студента

```
Student = namedtuple('Student', ['id', 'name', 'age', 'grades', 'active'],  
                    defaults=[[], True])
```

Створюємо список студентів

```
students = [  
    Student(1, 'Олена Петренко', 19, [85, 90, 78, 92]),  
    Student(2, 'Іван Коваленко', 20, [75, 88, 95, 70]),  
    Student(3, 'Марія Шевченко', 18, [95, 92, 88, 96]),  
    Student(4, 'Петро Бондаренко', 21, [60, 75, 80, 65]),  
    Student(5, 'Анна Кравченко', 20, [100, 98, 95, 99])  
]
```

```
print("📖 БАЗА ДАНИХ СТУДЕНТІВ (namedtuple)")
```

```

print("=" * 70)

# Функції для роботи з даними
def average_grade(student):
    """Обчислює середній бал студента"""
    return sum(student.grades) / len(student.grades) if student.grades else 0

def student_info(student):
    """Повертає інформацію про студента"""
    avg = average_grade(student)
    status = "активний" if student.active else "неактивний"
    return f"{student.name} (ID:{student.id}), {student.age} років, сер.бал: {avg:.2f}, {status}"

# Виводимо інформацію
print("Список студентів:")
for student in students:
    print(f" {student_info(student)}")

# Знаходимо найкращого студента
best_student = max(students, key=average_grade)
print(f"\n🏆 Найкращий студент: {best_student.name} (сер.бал: {average_grade(best_student):.2f})")

# Фільтруємо студентів за віком
young_students = [s for s in students if s.age <= 19]
print(f"\n👶 Молодші студенти (<=19): {len(young_students)}")

# Додаємо оцінку (створюємо нового студента через _replace)
updated_students = []
for student in students:
    new_grades = student.grades + [random.randint(70, 100)]
    updated_students.append(student._replace(grades=new_grades))

print(f"\n✅ Додано нові оцінки для всіх студентів")

```

Порівняння namedtuple з іншими структурами

```
from collections import namedtuple
import sys
```

1. Звичайний кортеж

```
tuple_point = (10, 20)
print(f'Кортеж: {tuple_point}, розмір: {sys.getsizeof(tuple_point)} байт")
# Доступ: tuple_point[0] – незрозуміло що це
```

2. Словник

```
dict_point = {'x': 10, 'y': 20}
print(f'Словник: {dict_point}, розмір: {sys.getsizeof(dict_point)} байт")
# Доступ: dict_point['x'] – зрозуміло, але більше пам'яті
```

3. namedtuple

```
NamedPoint = namedtuple('NamedPoint', ['x', 'y'])
named_point = NamedPoint(10, 20)
print(f"NamedTuple: {named_point}, розмір: {sys.getsizeof(named_point)}
байт")
# Доступ: named_point.x – і зрозуміло, і економно
```

4. Клас

```
class ClassPoint:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class_point = ClassPoint(10, 20)
print(f'Клас: {class_point}, розмір: {sys.getsizeof(class_point)} байт")
# Доступ: class_point.x – зрозуміло, але найбільше пам'яті
```

2. defaultdict та Counter: словники з надздібностями

defaultdict – словник зі значенням за замовчуванням

Звичайний словник генерує помилку `KeyError`, якщо звернутися до відсутнього ключа. `defaultdict` автоматично створює значення для нового ключа, використовуючи фабрику за замовчуванням.

```
from collections import defaultdict

# Звичайний словник – проблема
regular_dict = {}
# regular_dict['key'] # KeyError!

# defaultdict з фабрикою int (повертає 0)
dd_int = defaultdict(int)
print(dd_int['key']) # 0 – ключ автоматично створено!
dd_int['key'] += 5
print(dd_int) # defaultdict(<class 'int'>, {'key': 5})

# Різні фабрики за замовчуванням
dd_list = defaultdict(list) # для списків
dd_dict = defaultdict(dict) # для словників
dd_set = defaultdict(set)   # для множин
dd_str = defaultdict(str)   # для рядків (порожній рядок)
dd_float = defaultdict(float) # для чисел з плаваючою комою
dd_lambda = defaultdict(lambda: 'немає') # власна функція

print(dd_list['key']) # []
print(dd_dict['key']) # {}
print(dd_set['key'])  # set()
print(dd_str['key'])  # ''
print(dd_float['key']) # 0.0
print(dd_lambda['key']) # 'немає'
```

Чому defaultdict корисний?

Розглянемо задачу групування елементів. Без `defaultdict` це виглядає так:

```
# Задача: згрупувати слова за їх першою літерою
words = ['яблуко', 'банан', 'апельсин', 'груша', 'абрикос', 'вишня',
'баклажан']
```

Без defaultdict

```
groups = {}
for word in words:
    first = word[0]
    if first not in groups:
        groups[first] = []
    groups[first].append(word)

print("Без defaultdict:")
for letter, word_list in sorted(groups.items()):
    print(f" {letter}: {word_list}")
```

З defaultdict

```
from collections import defaultdict
groups = defaultdict(list)
for word in words:
    groups[word[0]].append(word) # список автоматично створюється!

print("\nЗ defaultdict:")
for letter, word_list in sorted(groups.items()):
    print(f" {letter}: {word_list}")
```

Ще приклади використання defaultdict

Підрахунок кількості слів

```
text = "яблуко банан апельсин яблуко груша банан яблуко апельсин банан  
яблуко"
```

```
words = text.split()
```

Без defaultdict

```
counts = {}
for word in words:
    if word in counts:
        counts[word] += 1
    else:
        counts[word] = 1
```

З defaultdict

```

counts = defaultdict(int)
for word in words:
    counts[word] += 1

print("Частота слів:")
for word, count in sorted(counts.items()):
    print(f" {word}: {count}")

```

Групування чисел за парністю

```

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
groups = defaultdict(list)

```

```

for num in numbers:
    groups['парні' if num % 2 == 0 else 'непарні'].append(num)

```

```

print("\nГрупування чисел:")
for key, values in groups.items():
    print(f" {key}: {values}")

```

Counter – спеціалізований лічильник

Counter – це підклас словника, призначений для підрахунку об'єктів. Він має багато корисних методів.

```

from collections import Counter

```

Створення Counter

З послідовності

```

words = ['яблуко', 'банан', 'яблуко', 'апельсин', 'яблуко', 'банан']
counter1 = Counter(words)
print(f"Counter зі списку: {counter1}")

```

З рядка

```

text = "hello world hello python hello world"
counter2 = Counter(text.split())
print(f"Counter з рядка: {counter2}")

```

Зі словника

```

counter3 = Counter({'apple': 3, 'banana': 2, 'orange': 1})

```

```
print(f"Counter зі словника: {counter3}")
```

```
# 3 іменованими аргументами
```

```
counter4 = Counter(apple=3, banana=2, orange=1)
```

```
print(f"Counter з аргументів: {counter4}")
```

Основні методи Counter

```
from collections import Counter
```

```
# Створюємо лічильник
```

```
sentence = "python is awesome and python is powerful and python is fun"
```

```
word_counts = Counter(sentence.split())
```

```
print(f"Повний лічильник: {word_counts}")
```

```
print(f"Кількість слів: {sum(word_counts.values())}")
```

```
# most_common() – найчастіші елементи
```

```
print("\nНайчастіші слова:")
```

```
for word, count in word_counts.most_common(3):
```

```
    print(f" {word}: {count}")
```

```
# Доступ до елементів
```

```
print(f"\nКількість 'python': {word_counts['python']}")
```

```
print(f"Кількість 'java' (немає): {word_counts['java']}") # 0, не помилка!
```

```
# elements() – повертає елементи з повтореннями
```

```
counter = Counter(a=3, b=2, c=1)
```

```
print(f"\nelements(): {list(counter.elements())}") # ['a', 'a', 'a', 'b', 'b', 'c']
```

```
# Оновлення лічильника
```

```
word_counts.update(['python', 'python', 'java'])
```

```
print(f"\nПісля оновлення: {word_counts['python']}, {word_counts['java']}")
```

```
# Віднімання
```

```
word_counts.subtract(['python', 'python'])
```

```
print(f"Після віднімання: {word_counts['python']}")
```

Арифметичні операції з Counter

```
from collections import Counter
```

```
# Створюємо два лічильники
```

```
counter_a = Counter(a=3, b=2, c=1)
```

```
counter_b = Counter(a=1, b=2, c=3, d=1)
```

```
print(f"A: {counter_a}")
```

```
print(f"B: {counter_b}")
```

```
# Додавання
```

```
print(f"A + B: {counter_a + counter_b}")
```

```
# Віднімання (залишає тільки додатні)
```

```
print(f"A - B: {counter_a - counter_b}")
```

```
# Перетин (мінімум)
```

```
print(f"A & B: {counter_a & counter_b}")
```

```
# Об'єднання (максимум)
```

```
print(f"A | B: {counter_a | counter_b}")
```

```
# Порівняння
```

```
print(f"A == Counter(a=3,b=2,c=1)? {counter_a == Counter(a=3, b=2, c=1)}")
```

Практичний приклад: Аналіз тексту

```
from collections import Counter, defaultdict
```

```
import re
```

```
def analyze_text(text):
```

```
    """Комплексний аналіз тексту з використанням Counter та defaultdict"""
```

```
    print("██ АНАЛІЗ ТЕКСТУ")
```

```
    print("=" * 70)
```

```
# Очищуємо текст і розбиваємо на слова
```



```
words = re.findall(r'\b\w+\b', text.lower())
```

1. Загальна статистика

```
word_count = len(words)
```

```
unique_words = len(set(words))
```

```
print(f"Загальна кількість слів: {word_count}")
```

```
print(f"Унікальних слів: {unique_words}")
```

```
print(f"Лексичне багатство: {unique_words/word_count:.2%}")
```

2. Найчастіші слова (Counter)

```
word_freq = Counter(words)
```

```
print("\n📊 Топ-10 найчастіших слів:")
```

```
for word, count in word_freq.most_common(10):
```

```
    print(f" {word}: {count} разів ({count/word_count*100:.1f}%)")
```

3. Розподіл слів за довжиною (defaultdict)

```
length_dist = defaultdict(list)
```

```
for word in words:
```

```
    length_dist[len(word)].append(word)
```

```
print("\n📏 Розподіл слів за довжиною:")
```

```
for length in sorted(length_dist.keys()):
```

```
    words_of_length = length_dist[length]
```

```
    print(f" {length} літер: {len(words_of_length)} слів")
```

```
    if length <= 5: # показуємо приклади для коротких слів
```

```
        examples = set(words_of_length[:3])
```

```
        print(f"   приклади: {', '.join(examples)}")
```

4. Слова, які починаються з кожної літери

```
first_letter = defaultdict(list)
```

```
for word in words:
```

```
    first_letter[word[0]].append(word)
```

```
print("\n🔤 Слова за першою літерою:")
```

```
for letter in sorted(first_letter.keys()):
```

```
    count = len(set(first_letter[letter]))
```

```
print(f" {letter}: {count} унікальних слів")
```

```
# 5. Перевірка, чи є слова паліндромами
```

```
palindromes = [word for word in set(words) if word == word[::-1] and  
len(word) > 1]
```

```
print(f"\n🌀 Знайдені паліндроми: {', '.join(palindromes) if palindromes  
else 'немає'}")
```

```
return word_freq
```

```
# Тестуємо
```

```
sample_text = """
```

Python is an interpreted, high-level, general-purpose programming language.

Created by Guido van Rossum and first released in 1991, Python's design

philosophy emphasizes code readability with its notable use of significant

whitespace. Its language constructs and object-oriented approach aim to help

programmers write clear, logical code for small and large-scale projects.

Python is dynamically typed and garbage-collected. It supports multiple

programming paradigms, including procedural, object-oriented, and functional

programming. Python is often described as a "batteries included" language

due to its comprehensive standard library.

```
"""
```

```
freq = analyze_text(sample_text)
```

Практичний приклад: Система голосування

```
from collections import Counter, defaultdict
```

```
class VotingSystem:
```

```
    """Система для проведення голосувань"""
```

```
    def __init__(self):
```

```
        self.votes = defaultdict(list) # кандидат -> список виборців
```

```
        self.voters = set() # хто вже проголосував
```

```
    def vote(self, voter_name, candidate):
```

```
        """Додає голос виборця за кандидата"""
```

```

if voter_name in self.voters:
    return f"✗ {voter_name}, ви вже голосували!"

self.votes[candidate].append(voter_name)
self.voters.add(voter_name)
return f"✓ {voter_name} проголосував за {candidate}"

def get_results(self):
    """Повертає результати голосування"""
    # Підраховуємо голоси за допомогою Counter
    vote_counts = Counter()
    for candidate, voters in self.votes.items():
        vote_counts[candidate] = len(voters)

    return vote_counts

def get_winner(self):
    """Визначає переможця"""
    results = self.get_results()
    if not results:
        return "Голосування ще не проводилось"

    # Знаходимо максимальну кількість голосів
    max_votes = max(results.values())
    winners = [c for c, v in results.items() if v == max_votes]

    if len(winners) == 1:
        return f"🏆 Переможець: {winners[0]} з {max_votes} голосами"
    else:
        return f"🗳 Нічия між {', '.join(winners)} ({max_votes} голосів)"

def get_voters_for(self, candidate):
    """Повертає список виборців за кандидата"""
    return self.votes.get(candidate, [])

def summary(self):
    """Повна статистика"""

```

```
results = self.get_results()
```

```
print("\n📊 РЕЗУЛЬТАТИ ГОЛОСУВАННЯ")
```

```
print("=" * 50)
```

```
print(f"Всього виборців: {len(self.voters)}")
```

```
print("\nКандидати:")
```

```
for candidate, count in results.most_common():
```

```
    percentage = (count / len(self.voters)) * 100 if self.voters else 0
```

```
    voters_list = ', '.join(self.get_voters_for(candidate))
```

```
    print(f" {candidate}: {count} голосів ({percentage:.1f}%)")
```

```
    print(f"   голосували: {voters_list}")
```

```
print(f"\n{self.get_winner()}")
```

Проводимо голосування

```
vote_system = VotingSystem()
```

```
print("🗳️ СИСТЕМА ГОЛОСУВАННЯ")
```

```
print("=" * 60)
```

Голосуємо

```
print(vote_system.vote("Олена", "Кандидат А"))
```

```
print(vote_system.vote("Іван", "Кандидат Б"))
```

```
print(vote_system.vote("Марія", "Кандидат А"))
```

```
print(vote_system.vote("Петро", "Кандидат Б"))
```

```
print(vote_system.vote("Анна", "Кандидат А"))
```

```
print(vote_system.vote("Олена", "Кандидат Б")) # спроба голосувати вдруге
```

```
print(vote_system.vote("Дмитро", "Кандидат Б"))
```

```
print(vote_system.vote("Катерина", "Кандидат В"))
```

Результати

```
vote_system.summary()
```

Практичний приклад: Аналіз логів веб-сервера

```
from collections import Counter, defaultdict
```

```

import re

def analyze_server_logs(log_lines):
    """
    Аналізує логи веб-сервера.
    Формат логу: IP - - [дата] "METHOD URL PROTOCOL" STATUS SIZE
    """

    # Структури даних для зберігання статистики
    ip_counter = Counter() # лічильник запитів за IP
    url_counter = Counter() # лічильник запитів за URL
    status_counter = Counter() # лічильник статус-кодів
    hourly_stats = defaultdict(lambda: {'count': 0, 'ips': set()}) # статистика
за годинами

    # Регулярний вираз для парсингу логів (спрощений)
    pattern = r'(\d+\.\d+\.\d+\.\d+).*?[(.*?):(\d+):.*?] "(?:GET|POST) (.*) HTTP.*?" (\d+)'

    for line in log_lines:
        match = re.search(pattern, line)
        if not match:
            continue

        ip, date, hour, url, status = match.groups()

        # Оновлюємо лічильники
        ip_counter[ip] += 1
        url_counter[url] += 1
        status_counter[int(status)] += 1
        hourly_stats[hour]['count'] += 1
        hourly_stats[hour]['ips'].add(ip)

    # Виводимо результати
    print("\nАНАЛІЗ ЛОГІВ СЕРВЕРА")
    print("=" * 70)

```

```

print(f"\n🔍 Загальна статистика:")
print(f" Всього запитів: {sum(ip_counter.values())}")
print(f" Унікальних IP: {len(ip_counter)}")
print(f" Унікальних URL: {len(url_counter)}")

print(f"\n📊 Топ-5 IP за кількістю запитів:")
for ip, count in ip_counter.most_common(5):
    print(f" {ip}: {count} запитів")

print(f"\n🌐 Топ-5 популярних URL:")
for url, count in url_counter.most_common(5):
    print(f" {url}: {count} запитів")

print(f"\n📈 Розподіл статус-кодів:")
for status in sorted(status_counter.keys()):
    count = status_counter[status]
    percentage = (count / sum(status_counter.values())) * 100
    print(f" {status}: {count} ({percentage:.1f}%)")

print(f"\n🕒 Активність за годинами:")
for hour in sorted(hourly_stats.keys()):
    stats = hourly_stats[hour]
    print(f" {hour}:00 — {stats['count']} запитів, {len(stats['ips'])} унікальних IP")

return {
    'ip': ip_counter,
    'url': url_counter,
    'status': status_counter,
    'hourly': hourly_stats
}

# Тестові логи
test_logs = [
    '192.168.1.100 - - [10/Oct/2023:13:55:36 +0000] "GET /index.html HTTP/1.1" 200 2326',
    '192.168.1.101 - - [10/Oct/2023:13:56:12 +0000] "GET /about.html HTTP/1.1" 200 1542',

```

```

'192.168.1.100 - - [10/Oct/2023:13:57:23 +0000] "GET /style.css HTTP/1.1" 200 1245',
'192.168.1.102 - - [10/Oct/2023:13:58:45 +0000] "POST /login HTTP/1.1" 302 0',
'192.168.1.100 - - [10/Oct/2023:14:01:32 +0000] "GET /dashboard HTTP/1.1" 403 234',
'192.168.1.103 - - [10/Oct/2023:14:02:54 +0000] "GET /index.html HTTP/1.1" 200 2326',
'192.168.1.101 - - [10/Oct/2023:14:03:17 +0000] "GET /products.html HTTP/1.1" 200 3421',
'192.168.1.104 - - [10/Oct/2023:14:05:43 +0000] "GET /index.html HTTP/1.1" 404 123',
'192.168.1.100 - - [10/Oct/2023:14:06:28 +0000] "GET /api/data HTTP/1.1" 200 567',
'192.168.1.102 - - [10/Oct/2023:14:07:15 +0000] "GET /logout HTTP/1.1" 200 0',
]

```

```
analyze_server_logs(test_logs)
```

У наступному пункті ми розглянемо інші корисні структури з модуля `collections`: `OrderedDict`, `ChainMap`, а також заглибимося в `comprehensions` та генератори.

3. `OrderedDict`, `ChainMap` та інші корисні структури

Модуль `collections` містить ще кілька дуже корисних структур, які варто знати. Хоча деякі з них у сучасних версіях Python (3.7+) стали менш актуальними через зміни в самій мові, вони все ще корисні для розуміння та для роботи зі старим кодом.

`OrderedDict` – словник, що пам'ятає порядок додавання

До Python 3.7 звичайні словники не зберігали порядок додавання елементів. `OrderedDict` був створений саме для цього. У сучасному Python звичайний `dict` також зберігає порядок, але `OrderedDict` має деякі додаткові можливості.

```
from collections import OrderedDict
```

```
# Створення OrderedDict
```

```
ordered = OrderedDict()
```

```
ordered['banana'] = 3
```

```
ordered['apple'] = 5
```

```
ordered['orange'] = 2
```

```
ordered['grape'] = 4
```

```
print("\n3.2 ORDEREDDICT")
```

```

print("=" * 60)
print(f"Вміст: {ordered}")
print(f"Ключі в порядку додавання: {list(ordered.keys())}")

# Порівняння зі звичайним словником
regular_dict = {}
regular_dict['banana'] = 3
regular_dict['apple'] = 5
regular_dict['orange'] = 2
regular_dict['grape'] = 4

print(f"\nЗвичайний dict: {regular_dict}")
print(f"Чи рівні? {ordered == regular_dict}")

# Специфічні методи OrderedDict
print("\n☞ Спеціальні методи OrderedDict:")

# move_to_end – переміщує елемент у кінець
ordered.move_to_end('banana')
print(f"Після move_to_end('banana'): {list(ordered.keys())}")

# move_to_end з last=False – переміщує на початок
ordered.move_to_end('orange', last=False)
print(f"Після move_to_end('orange', last=False): {list(ordered.keys())}")

# popitem(last=True) – видаляє останній елемент
item = ordered.popitem()
print(f"Видалено останній: {item}, тепер: {list(ordered.keys())}")

# popitem(last=False) – видаляє перший елемент
item = ordered.popitem(last=False)
print(f"Видалено перший: {item}, тепер: {list(ordered.keys())}")

```

Коли використовувати OrderedDict?

- Коли вам потрібно часто переміщувати елементи на початок або кінець
- Коли порядок ключів має критичне значення для логіки програми
- Для сумісності зі старим кодом (до Python 3.7)

ChainMap – об'єднання кількох словників

ChainMap групує кілька словників в єдине відображення. Пошук ключа відбувається послідовно в усіх словниках. Це корисно для роботи з областями видимості, конфігураціями за замовчуванням тощо.

```
from collections import ChainMap

print("↔ CHAINMAP")
print("=" * 60)

# Створюємо кілька словників з налаштуваннями
defaults = {
    'theme': 'light',
    'language': 'uk',
    'font_size': 12,
    'notifications': True
}

user_prefs = {
    'theme': 'dark',
    'font_size': 14,
    'auto_save': True
}

session_prefs = {
    'language': 'en'
}

# Об'єднуємо їх у ChainMap (пріоритет: перший словник)
config = ChainMap(session_prefs, user_prefs, defaults)

print(f"Повна конфігурація:")
for key, value in config.items():
    print(f" {key}: {value}")

print(f"\nЗначення 'theme': {config['theme']} (з user_prefs)")
print(f"Значення 'language': {config['language']} (з session_prefs)")
```

```
print(f"Значення 'font_size': {config['font_size']} (з user_prefs)")
print(f"Значення 'notifications': {config['notifications']} (з defaults)")
```

Додавання нового словника

```
runtime_prefs = {'debug': True}
config = config.new_child(runtime_prefs)
print(f"\nПісля додавання runtime_prefs:")
print(f" debug: {config['debug']}")
```

Зміна значення – змінюється тільки в першому словнику

```
config['theme'] = 'high-contrast'
print(f"\nПісля зміни 'theme':")
print(f" session_prefs['theme']? {'theme' in session_prefs}")
print(f" user_prefs['theme'] тепер: {user_prefs['theme']}")
```

maps – список всіх словників

```
print(f"\nСтруктура ChainMap (3 словники):")
for i, mapping in enumerate(config.maps):
    print(f" Рівень {i}: {mapping}")
```

Практичний приклад: Система конфігурації додатку

```
from collections import ChainMap
import json
```

```
class ConfigManager:
```

```
    """Менеджер конфігурації з підтримкою рівнів налаштувань"""
```

```
    def __init__(self):
```

```
        # Рівні конфігурації (від найнижчого пріоритету до найвищого)
```

```
        self.defaults = {
            'host': 'localhost',
            'port': 8080,
            'debug': False,
            'database': 'sqlite:///app.db',
            'max_connections': 10,
            'timeout': 30,
            'language': 'uk',
```

```

        'theme': 'light'
    }

    self.file_config = {} # налаштування з файлу
    self.env_config = {} # налаштування з змінних середовища
    self.runtime_config = {} # налаштування, задані під час роботи

    # Створюємо ChainMap
    self._update_chain()

def _update_chain(self):
    """Оновлює ChainMap відповідно до поточних налаштувань"""
    self.config = ChainMap(
        self.runtime_config,
        self.env_config,
        self.file_config,
        self.defaults
    )

def load_from_file(self, filename):
    """Завантажує налаштування з JSON-файлу"""
    try:
        with open(filename, 'r', encoding='utf-8') as f:
            self.file_config = json.load(f)
            self._update_chain()
            print(f"✓ Завантажено налаштування з {filename}")
    except FileNotFoundError:
        print(f"✗ Файл {filename} не знайдено")
    except json.JSONDecodeError:
        print(f"✗ Помилка парсингу JSON у {filename}")

def load_from_env(self, env_vars):
    """Завантажує налаштування зі словника (імітація змінних
середовища)"""
    self.env_config = env_vars
    self._update_chain()
    print(f"✓ Завантажено {len(env_vars)} налаштувань з середовища")

```

```

def set_runtime(self, key, value):
    """Встановлює налаштування під час виконання"""
    self.runtime_config[key] = value
    self._update_chain()
    print(f"✓ Встановлено runtime: {key} = {value}")

def get(self, key, default=None):
    """Отримує значення налаштування"""
    return self.config.get(key, default)

def show_all(self):
    """Показує всі налаштування з джерелами"""
    print("\n📁 ПОТОЧНА КОНФІГУРАЦІЯ:")
    print("=" * 60)

    # Збираємо всі ключі
    all_keys = set()
    for mapping in self.config.maps:
        all_keys.update(mapping.keys())

    for key in sorted(all_keys):
        value = self.config[key]
        # Визначаємо джерело
        source = None
        for mapping in self.config.maps:
            if key in mapping:
                if mapping is self.runtime_config:
                    source = "runtime"
                elif mapping is self.env_config:
                    source = "env"
                elif mapping is self.file_config:
                    source = "file"
                else:
                    source = "default"
                break

```

```

    print(f" {key:20} = {str(value):20} [{source}]")

# Демонстрація
print("❖ СИСТЕМА КОНФІГУРАЦІЇ")
print("=" * 60)

config = ConfigManager()

# Імітуємо завантаження з файлу
config.load_from_file('config.json') # файл не існує – ігноруємо

# Імітуємо змінні середовища
env_vars = {
    'port': 3000,
    'debug': 'true',
    'database': 'postgresql://localhost/mydb'
}
config.load_from_env(env_vars)

# Встановлюємо runtime налаштування
config.set_runtime('debug', True)
config.set_runtime('theme', 'dark')

# Показуємо результати
config.show_all()

print(f"\n❖ Отримання окремих значень:")
print(f" host: {config.get('host')}")
print(f" port: {config.get('port')}")
print(f" debug: {config.get('debug')}")
print(f" theme: {config.get('theme')}")
print(f" non_existent: {config.get('non_existent', 'default_value')}")

```

Інші корисні структури

```

from collections import deque, UserDict, UserList, UserString

```

```
# deque ми вже розглядали в попередній лекції
print("\n🔄 DEQUE (ще раз про корисні методи)")
d = deque([1, 2, 3, 4, 5], maxlen=10)
d.rotate(2) # ротація
print(f"Після rotate(2): {list(d)}")
```

```
# UserDict, UserList, UserString – для створення власних класів
from collections import UserDict
```

```
class CaseInsensitiveDict(UserDict):
    """Словник, де ключі нечутливі до регістру"""

    def __setitem__(self, key, value):
        super().__setitem__(key.lower(), value)

    def __getitem__(self, key):
        return super().__getitem__(key.lower())

    def __contains__(self, key):
        return super().__contains__(key.lower())
```

```
# Демонстрація
ci_dict = CaseInsensitiveDict()
ci_dict['Name'] = 'Олена'
ci_dict['Age'] = 19

print(f"\n📁 UserDict (нечутливий до регістру):")
print(f" name: {ci_dict['name']}") # працює!
print(f" NAME: {ci_dict['NAME']}") # працює!
print(f" Вміст: {ci_dict.data}")
```

4. Вибір оптимальної структури даних та list comprehensions

Після вивчення всіх структур даних постає важливе питання: яку структуру вибрати для конкретної задачі? Правильний вибір може значно підвищити ефективність та читабельність коду.

Порівняння швидкості різних структур

```
import time
import sys
from collections import deque

def compare_performance():
    """Порівнює швидкість різних операцій на різних структурах"""

    print("✂ ПОРІВНЯННЯ ПРОДУКТИВНОСТІ")
    print("=" * 70)

    size = 100000

    # 1. Пошук елемента
    print("\n🔍 ПОШУК ЕЛЕМЕНТА:")

    # Список
    lst = list(range(size))
    target = size - 1

    start = time.time()
    result = target in lst
    end = time.time()
    print(f"Список (in): {end-start:.6f} сек")

    # Множина
    st = set(range(size))
    start = time.time()
    result = target in st
    end = time.time()
    print(f"Множина (in): {end-start:.6f} сек")

    # Словник
    dct = {i: i for i in range(size)}
    start = time.time()
    result = target in dct
```

```

end = time.time()
print(f" Словник (in): {end-start:.6f} сек")

# 2. Додавання елементів
print("\n⬇ ДОДАВАННЯ ЕЛЕМЕНТІВ:")

# Список (в кінець)
lst = []
start = time.time()
for i in range(size):
    lst.append(i)
end = time.time()
print(f" Список (append): {end-start:.4f} сек")

# Список (на початок) – дуже повільно!
lst = []
start = time.time()
for i in range(size):
    lst.insert(0, i)
end = time.time()
print(f" Список (insert 0): {end-start:.4f} сек")

# Deque (на початок) – швидко!
dq = deque()
start = time.time()
for i in range(size):
    dq.appendleft(i)
end = time.time()
print(f" Deque (appendleft): {end-start:.4f} сек")

# 3. Видалення елементів
print("\n🗑 ВИДАЛЕННЯ ЕЛЕМЕНТІВ:")

# Список (з кінця)
lst = list(range(size))
start = time.time()
for i in range(size):

```



```
lst.pop()
end = time.time()
print(f" Список (pop з кінця): {end-start:.4f} сек")
```

Список (з початку) – повільно!

```
lst = list(range(size))
start = time.time()
for i in range(size):
    lst.pop(0)
end = time.time()
print(f" Список (pop 0): {end-start:.4f} сек")
```

Deque (з початку) – швидко!

```
dq = deque(range(size))
start = time.time()
for i in range(size):
    dq.popleft()
end = time.time()
print(f" Deque (popleft): {end-start:.4f} сек")
```

Порівнюємо

```
compare_performance()
```

Таблиця вибору структур даних

Задача	Рекомендована структура	Чому
Збереження порядку елементів	list	Впорядкована, змінна
Частий пошук за значенням	set	O(1) пошук
Унікальність елементів	set	Автоматично видаляє дублікати
Пари ключ-значення	dict	Швидкий доступ за ключем

Задача	Рекомендована структура	Чому
Часті вставки/видалення з обох кінців	deque	O(1) з обох кінців
Прості структури з іменованими полями	namedtuple	Легкі, незмінні, читабельні
Лічильник елементів	Counter	Спеціалізовані методи
Автоматичне створення ключів	defaultdict	Уникає KeyError
Об'єднання кількох словників	ChainMap	Не створює копій
Незмінні дані	tuple	Гарантує незмінність

List Comprehensions – елегантне створення списків

List comprehensions – це один з найпопулярніших "пітонічних" прийомів. Вони дозволяють створювати списки коротко та виразно.

```
print("📦 LIST COMPREHENSIONS")
print("=" * 70)
```

```
# Базова форма: [вираз for елемент in послідовність]
squares = [x**2 for x in range(10)]
print(f"Квадрати чисел: {squares}")
```

```
# З умовою: [вираз for елемент in послідовність if умова]
even_squares = [x**2 for x in range(20) if x % 2 == 0]
print(f"Квадрати парних чисел: {even_squares}")
```

```
# З вкладеними циклами
pairs = [(x, y) for x in range(3) for y in range(2)]
print(f"Всі пари: {pairs}")
```

```
# З умовою на вираз (тернарний оператор)
```

```

parity = ["парне" if x % 2 == 0 else "непарне" for x in range(10)]
print(f'Парність: {parity}')

# Зі складною обробкою
words = ["яблуко", "банан", "апельсин", "груша", "слива"]
word_lengths = [(word, len(word)) for word in words if len(word) > 5]
print(f'Довгі слова: {word_lengths}')

```

Порівняння з циклом

Без comprehension

```

result1 = []
for i in range(10):
    if i % 2 == 0:
        result1.append(i**2)

```

3 comprehension

```

result2 = [i**2 for i in range(10) if i % 2 == 0]

```

```

print(f'Результат однаковий: {result1 == result2}')

```

Dict Comprehensions

```

print("\n📖 DICT COMPREHENSIONS")
print("=" * 70)

```

Квадрати чисел як словник

```

squares_dict = {x: x**2 for x in range(1, 6)}
print(f'Квадрати: {squares_dict}')

```

З умовою

```

even_squares_dict = {x: x**2 for x in range(1, 11) if x % 2 == 0}
print(f'Квадрати парних: {even_squares_dict}')

```

Інвертування словника

```

original = {'a': 1, 'b': 2, 'c': 3}
inverted = {value: key for key, value in original.items()}
print(f'Інвертований: {inverted}')

```

З двох списків

```
names = ['Олена', 'Іван', 'Марія']
ages = [19, 20, 18]
students = {name: age for name, age in zip(names, ages)}
print(f"Студенти: {students}")
```

З трансформацією

```
text = "hello world"
char_counts = {char: text.count(char) for char in set(text)}
print(f"Частота символів: {char_counts}")
```

Set Comprehensions

```
print("\n🔧 SET COMPREHENSIONS")
print("=" * 70)
```

Множина квадратів

```
squares_set = {x**2 for x in range(-5, 5)}
print(f"Квадрати (унікальні): {squares_set}")
```

Унікальні літери в тексті

```
text = "hello world hello python"
unique_chars = {char for char in text if char.isalpha()}
print(f"Унікальні літери: {sorted(unique_chars)}")
```

Довжини слів (унікальні)

```
words = ["яблуко", "банан", "апельсин", "груша", "ананас"]
word_lengths_set = {len(word) for word in words}
print(f"Унікальні довжини слів: {word_lengths_set}")
```

Generator Expressions

Генератори схожі на list comprehensions, але не створюють весь список у пам'яті, а генерують елементи "на льоту". Це важливо для великих даних.

```
print("\n🔧 GENERATOR EXPRESSIONS")
print("=" * 70)
```

List comprehension – створює весь список

```

list_squares = [x**2 for x in range(10)]
print(f"List comprehension: {list_squares}, розмір:
{sys.getsizeof(list_squares)} байт")

# Generator expression – створює генератор
gen_squares = (x**2 for x in range(10))
print(f"Generator expression: {gen_squares}, розмір:
{sys.getsizeof(gen_squares)} байт")

# Використання генератора
print("Елементи з генератора:", end=" ")
for val in gen_squares:
    print(val, end=" ")
print()

# Для великих даних різниця величезна!
big_range = 1000000
list_big = [x for x in range(big_range)]
gen_big = (x for x in range(big_range))

print(f"\nРозмір списку з {big_range} елементів: {sys.getsizeof(list_big)}
байт")
print(f"Розмір генератора з {big_range} елементів: {sys.getsizeof(gen_big)}
байт")

# Коли використовувати генератори:
# 1. Для дуже великих даних
# 2. Коли не потрібно зберігати всі значення одночасно
# 3. Для ланцюжків обробки даних

# Приклад: сума квадратів перших 10 мільйонів чисел
# Генератор не створює весь список у пам'яті!
sum_squares = sum(x**2 for x in range(10000000))
print(f"\nСума квадратів (без створення списку): {sum_squares}")

```

Практичний приклад: Вибір оптимальної структури для аналізу даних

```

from collections import Counter, defaultdict, namedtuple

```

```

import random
import time

# Визначаємо структуру для транзакції
Transaction = namedtuple('Transaction', ['id', 'user_id', 'amount', 'category',
'timestamp'])

def generate_transactions(n=10000):
    """Генерує тестові транзакції"""
    categories = ['food', 'transport', 'entertainment', 'utilities', 'shopping',
'health']
    transactions = []

    for i in range(n):
        transaction = Transaction(
            id=i,
            user_id=random.randint(1, 100),
            amount=round(random.uniform(10, 500), 2),
            category=random.choice(categories),
            timestamp=random.randint(1600000000, 1700000000)
        )
        transactions.append(transaction)

    return transactions

def analyze_transactions(transactions):
    """Аналізує транзакції, використовуючи різні структури даних"""

    print("■ АНАЛІЗ ТРАНЗАКЦІЙ")
    print("=" * 70)
    print(f"Всього транзакцій: {len(transactions)}")

    # 1. Підрахунок за категоріями (Counter)
    category_counter = Counter(t.category for t in transactions)
    print("\n■ Транзакції за категоріями:")
    for category, count in category_counter.most_common():
        total_amount = sum(t.amount for t in transactions if t.category == category)

```

```
print(f" {category:15} {count:6} транзакцій, загальна сума:
{total_amount:10.2f}")
```

2. Групування за користувачами (defaultdict)

```
user_transactions = defaultdict(list)
for t in transactions:
    user_transactions[t.user_id].append(t)

print("\n👤 Топ-5 користувачів за кількістю транзакцій:")
top_users = sorted(user_transactions.items(), key=lambda x: len(x[1]),
reverse=True)[:5]
for user_id, trans in top_users:
    total = sum(t.amount for t in trans)
    print(f" Користувач {user_id:3}: {len(trans)} транзакцій, загальна сума:
{total:8.2f}")
```

3. Статистика за місяцями (dict comprehension)

```
from datetime import datetime
months = {datetime.fromtimestamp(t.timestamp).strftime('%Y-%m')
           for t in transactions}

monthly_stats = {
    month: {
        'count': len([t for t in transactions if
datetime.fromtimestamp(t.timestamp).strftime('%Y-%m') == month]),
        'total': sum(t.amount for t in transactions if
datetime.fromtimestamp(t.timestamp).strftime('%Y-%m') == month)
    }
    for month in sorted(months)
}

print("\n📊 Статистика за місяцями:")
for month, stats in monthly_stats.items():
    print(f" {month}: {stats['count']} транзакцій, {stats['total']:.2f} грн")
```

4. Середній чек за категоріями (dict comprehension)

```
avg_check = {
```

```

        category: sum(t.amount for t in transactions if t.category == category) /
            len([t for t in transactions if t.category == category])
    for category in categories
}

print("\n💰 Середній чек за категоріями:")
for category, avg in sorted(avg_check.items(), key=lambda x: x[1],
reverse=True):
    print(f" {category:15}: {avg:8.2f} грн")

# 5. Пошук аномалій (list comprehension)
all_amounts = [t.amount for t in transactions]
mean = sum(all_amounts) / len(all_amounts)
std = (sum((x - mean) ** 2 for x in all_amounts) / len(all_amounts)) ** 0.5

anomalies = [t for t in transactions if abs(t.amount - mean) > 3 * std]

print(f"\n⚠️ Аномальні транзакції (більше 3σ): {len(anomalies)}")
for t in anomalies[:5]: # перші 5
    print(f" {t.category:15} {t.amount:8.2f} грн (користувач {t.user_id})")

# Генеруємо дані та аналізуємо
transactions = generate_transactions(10000)
analyze_transactions(transactions)

```

Підсумкові рекомендації

1. Для більшості задач використовуйте вбудовані типи: list, dict, set, tuple.
2. Обирайте структуру за операціями, які виконуєте найчастіше:
 - Частий пошук → set або dict
 - Часті вставки/видалення з кінця → list
 - Часті вставки/видалення з початку → deque
 - Підрахунок елементів → Counter
 - Автоматичне створення ключів → defaultdict
3. Для великих даних використовуйте генератори замість списків.
4. List comprehensions роблять код коротшим і читабельнішим, але не зловживайте ними, якщо вираз стає занадто складним.

5. Пам'ятайте про компроміси: швидкість vs пам'ять, простота vs ефективність.

У наступній лекції ми перейдемо до об'єктно-орієнтованого програмування – однієї з найважливіших парадигм у сучасній розробці.

Питання для обговорення

1. Що таке namedtuple і чим він кращий за звичайний кортеж? У яких ситуаціях його варто використовувати замість створення власного класу?

2. Проаналізуйте код:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(10, 20)
print(p.x, p[0])
```

Чому доступ до елементів можливий і через крапку, і через індекс? Яка фундаментальна властивість namedtuple це забезпечує?

3. Поясніть призначення defaultdict. Наведіть приклад задачі, де використання звичайного словника потребує постійної перевірки наявності ключа, а defaultdict робить код елегантнішим.

4. Напишіть програму, яка підраховує кількість кожного слова в тексті. Реалізуйте це двома способами: зі звичайним словником та з defaultdict(int). Порівняйте код.

5. Що таке Counter з модуля collections? Які методи цього класу дозволяють отримати найчастіші елементи? Наведіть приклад підрахунку голосів на виборах.

6. Поясніть різницю між Counter та звичайним словником. Які додаткові можливості надає Counter? Наприклад, що роблять методи most_common(), elements() та операції +, -, &, | між лічильниками?

7. Що таке OrderedDict? Чи потрібен він у сучасних версіях Python (3.7+), де звичайні словники зберігають порядок додавання? Коли OrderedDict все ще може бути корисним?

8. Поясніть призначення ChainMap. Як він об'єднує кілька словників? Наведіть практичний приклад використання ChainMap для роботи з конфігураціями (наприклад, пріоритет: аргументи командного рядка > змінні середовища > конфігураційний файл > значення за замовчуванням).

9. Проаналізуйте код:

```
from collections import deque
dq = deque([1, 2, 3])
dq.appendleft(0)
dq.append(4)
dq.rotate(2)
```

Що буде в dq після виконання цього коду? Для чого потрібен метод rotate()?

10. Порівняйте швидкість пошуку елемента в списку, множині та словнику. Чому множини та словники забезпечують пошук за $O(1)$, а списки – за $O(n)$? Яка ціна цієї швидкості?

ЛЕКЦІЯ 14

Класи та об'єкти. Основи ООП

План лекції

1. Вступ до ООП: від процедурного до об'єктно-орієнтованого мислення.

Пояснимо, чому виникла потреба в ООП, які проблеми воно вирішує. Розберемо основні принципи: інкапсуляцію, наслідування, поліморфізм. Пояснимо різницю між класом (шаблоном) та об'єктом (екземпляром) на життєвих прикладах.

2. Класи та об'єкти: створення власних типів даних.

Навчимося створювати класи за допомогою ключового слова `class`. Розберемо конструктор `__init__` та параметр `self`. Покажемо, як створювати атрибути (дані) та методи (дії) для об'єктів. Створимо наш перший клас і попрацюємо з його екземплярами.

3. Атрибути класу та екземпляра: де зберігати дані?

Пояснимо важливу відмінність між атрибутами класу (спільними для всіх об'єктів) та атрибутами екземпляра (унікальними для кожного об'єкта). Розберемо, як і коли використовувати кожен тип.

4. Методи класу, статичні методи та магічні методи.

Познайомимось з різними типами методів: звичайні методи екземпляра, методи класу (`@classmethod`) та статичні методи (`@staticmethod`). Розберемо, для чого вони потрібні. Також торкнемося магічних методів (`__str__`, `__repr__`), які роблять наші класи "розумнішими".

1. Вступ до ООП: від процедурного до об'єктно-орієнтованого мислення

Досі ми писали програми в **процедурному стилі**. Ми створювали функції, які оперували даними, і ці дані передавалися від однієї функції до іншої. Такий підхід добре працює для невеликих програм, але коли програма зростає, виникають проблеми.

Проблеми процедурного програмування

Уявімо, що ми пишемо програму для обліку студентів:

Процедурний підхід

```
def create_student(name, age, grades):  
    return {'name': name, 'age': age, 'grades': grades}
```

```
def add_grade(student, grade):
```

```

student['grades'].append(grade)

def average_grade(student):
    return sum(student['grades']) / len(student['grades'])

def student_info(student):
    return f"{student['name']}, {student['age']} років, сер.бал: {average_grade(student):.2f}"

# Використання
student1 = create_student('Олена', 19, [85, 90, 78])
add_grade(student1, 92)
print(student_info(student1))

```

Цей код працює, але має недоліки:

- **Дані та функції розділені** – важко бачити, які функції працюють з якими даними
- **Немає контролю доступу** – можна випадково змінити дані неправильно
- **Важко розширювати** – додавання нового типу даних (наприклад, викладача) вимагає створення нового набору функцій
- **Код стає заплутаним** – у великих програмах важко стежити за всіма функціями

Як ООП вирішує ці проблеми?

Об'єктно-орієнтоване програмування об'єднує **дані** та **методи** (функції для роботи з цими даними) в одне ціле – **об'єкт**. Це дозволяє моделювати реальний світ природніше.

Основні поняття ООП

1. **Клас (Class)** – це шаблон, за яким створюються об'єкти. Можна уявити клас як форму для випікання печива.
2. **Об'єкт (Object)** – це конкретний екземпляр класу. Це печиво, створене за формою.
3. **Атрибут (Attribute)** – це дані, які зберігаються в об'єкті (наприклад, ім'я студента).
4. **Метод (Method)** – це функція, яка належить класу і може працювати з його даними.

Три кити ООП

1. **Інкапсуляція (Encapsulation)** – об'єднання даних і методів в одному об'єкті та приховування внутрішньої реалізації. Користувач об'єкта не повинен знати, як він влаштований всередині, йому достатньо знати, які методи можна викликати.

2. **Наслідування (Inheritance)** – можливість створювати нові класи на основі існуючих, успадковуючи їх властивості та методи. Це дозволяє повторно використовувати код і створювати ієрархії класів.

3. **Поліморфізм (Polymorphism)** – можливість об'єктів різних класів відповідати на однакові виклики методів по-різному. Наприклад, і кішка, і собака можуть відповідати на метод speak(), але по-різному.

Аналогія з реальним світом

Уявімо, що ми хочемо описати автомобілі в програмі.

- **Клас** – це загальне поняття "Автомобіль". У класі описано, що будь-який автомобіль має колір, марку, швидкість (атрибути) і вміє їхати, гальмувати, сигналізувати (методи).

- **Об'єкти** – це конкретні автомобілі: "червоний Toyota мого сусіда", "синій BMW з автосалону". Кожен об'єкт має свої конкретні значення атрибутів.

- **Інкапсуляція** – щоб їхати на автомобілі, вам не потрібно знати, як влаштований двигун. Ви просто натискаєте педаль газу.

- **Наслідування** – можна створити клас "Електромобіль", який успадковує всі властивості звичайного автомобіля, але додає свої (зарядка батареї).

- **Поліморфізм** – і звичайний автомобіль, і електромобіль реагують на метод "заправитись", але по-різному.

Переваги ООП

- **Модульність** – код організований у логічні блоки (класи)
- **Повторне використання** – класи можна використовувати в різних програмах
- **Легкість підтримки** – зміни в одному класі не впливають на інші
- **Природне моделювання** – легше уявляти програму як набір взаємодіючих об'єктів

2. Класи та об'єкти: створення власних типів даних

Тепер перейдемо до практики і навчимося створювати власні класи в Python.

Синтаксис класу

```
class Student:
    """Простий клас для представлення студента"""
    pass # поки що порожній клас

# Створення об'єктів (екземплярів)
student1 = Student()
student2 = Student()

print(type(student1)) # <class '__main__.Student'>
print(isinstance(student1, Student)) # True
```

Додавання атрибутів

Атрибути – це дані, які зберігаються в об'єкті. Їх можна додавати безпосередньо:

```
class Student:
    pass

# Створюємо об'єкт
student = Student()

# Додаємо атрибути вручну
student.name = 'Олена'
student.age = 19
student.grades = [85, 90, 78]

print(f'Ім'я: {student.name}')
print(f'Вік: {student.age}')
print(f'Оцінки: {student.grades}')
```

Але такий підхід незручний – доводиться додавати атрибути вручну для кожного об'єкта.

Конструктор `__init__`

Спеціальний метод `__init__` (ініціалізатор) викликається автоматично при створенні об'єкта. Він дозволяє задати початкові значення атрибутів.

```
class Student:
    """Клас для представлення студента"""

    def __init__(self, name, age, grades=None):
        """Конструктор, викликається при створенні об'єкта"""
        print(f"Створюється студент {name}")
        self.name = name # self – посилання на сам об'єкт
        self.age = age
        self.grades = grades if grades else []

    def info(self):
        """Метод для виведення інформації про студента"""
        return f"{self.name}, {self.age} років, оцінок: {len(self.grades)}"

# Створюємо об'єкти
student1 = Student('Олена', 19, [85, 90, 78])
student2 = Student('Іван', 20) # grades буде []

print(student1.info())
print(student2.info())

# Кожен об'єкт має свої атрибути
print(f"student1.name: {student1.name}")
print(f"student2.name: {student2.name}")
```

Параметр self

`self` – це посилання на поточний екземпляр класу. Він обов'язково має бути першим параметром будь-якого методу. Назва `self` є конвенцією, але можна використовувати будь-яку назву (хоча краще дотримуватися стандарту).

```
class Point:
    def __init__(this, x, y): # можна назвати інакше, але не рекомендується
        this.x = x
        this.y = y
```

```
def distance_to_origin(self): # теж можна, але краще self
    return (self.x ** 2 + self.y ** 2) ** 0.5
```

```
p = Point(3, 4)
```

```
print(f"Відстань до початку координат: {p.distance_to_origin()}")
```

Додавання методів

Методи – це функції, які належать класу і працюють з його даними.

```
class Student:
```

```
    """Розширений клас студента з різними методами"""
```

```
    def __init__(self, name, age, grades=None):
```

```
        self.name = name
```

```
        self.age = age
```

```
        self.grades = grades if grades else []
```

```
    def add_grade(self, grade):
```

```
        """Додає оцінку студенту"""
```

```
        if 0 <= grade <= 100:
```

```
            self.grades.append(grade)
```

```
            print(f"✓ Оцінку {grade} додано студенту {self.name}")
```

```
        else:
```

```
            print(f"✗ Оцінка {grade} недійсна (має бути 0-100)")
```

```
    def average_grade(self):
```

```
        """Обчислює середній бал"""
```

```
        if not self.grades:
```

```
            return 0
```

```
        return sum(self.grades) / len(self.grades)
```

```
    def is_excellent(self):
```

```
        """Перевіряє, чи є студент відмінником (сер. бал >= 90)"""
```

```
        return self.average_grade() >= 90
```

```
    def info(self):
```

```
        """Повна інформація про студента"""
```



```

avg = self.average_grade()
status = "відмінник" if self.is_excellent() else "звичайний студент"
return f"{self.name}, {self.age} років, сер.бал: {avg:.2f} ({status})"

```

Демонстрація

```

student = Student('Марія', 18, [95, 92, 88])
print(student.info())

```

```

student.add_grade(96)
student.add_grade(150) # помилка
print(student.info())

```

```

print(f"Середній бал: {student.average_grade():.2f}")
print(f"Відмінник? {student.is_excellent()}")

```

Практичний приклад: клас BankAccount

```

class BankAccount:

```

```

    """Клас для представлення банківського рахунку"""

```

```

    def __init__(self, owner, initial_balance=0):
        self.owner = owner
        self.balance = initial_balance
        self.transactions = [] # історія транзакцій

```

```

    def deposit(self, amount):
        """Поповнення рахунку"""
        if amount <= 0:
            print("✗ Сума поповнення має бути додатною")
            return False

```

```

        self.balance += amount
        self.transactions.append(f"Поповнення: +{amount} грн")
        print(f"✓ Поповнено на {amount} грн. Поточний баланс: {self.balance}
грн")
        return True

```

```

def withdraw(self, amount):
    """Зняття коштів"""
    if amount <= 0:
        print("✗ Сума зняття має бути додатною")
        return False

    if amount > self.balance:
        print(f"✗ Недостатньо коштів. Доступно: {self.balance} грн")
        return False

    self.balance -= amount
    self.transactions.append(f"Зняття: -{amount} грн")
    print(f"✓ Знято {amount} грн. Поточний баланс: {self.balance} грн")
    return True

def transfer(self, other_account, amount):
    """Переказ коштів на інший рахунок"""
    if self.withdraw(amount):
        other_account.deposit(amount)
        self.transactions.append(f"Переказ:  -{amount}   грн   на   рахунок  
{other_account.owner}")
        other_account.transactions.append(f"Отримано переказ:  +{amount}  
грн від {self.owner}")
        print(f"✓ Переказ {amount} грн на рахунок {other_account.owner}  
виконано")
        return True
    return False

def show_transactions(self):
    """Показує історію транзакцій"""
    print(f"\n📁 Історія транзакцій для {self.owner}:")
    if not self.transactions:
        print("   Немає транзакцій")
    else:
        for i, trans in enumerate(self.transactions, 1):
            print(f"   {i}. {trans}")

```

```

def __str__(self):
    """Магічний метод для красивого виведення"""
    return f"Рахунок {self.owner}: {self.balance} грн"

# Демонстрація роботи
print("🏠 БАНКІВСЬКА СИСТЕМА")
print("=" * 60)

# Створюємо рахунки
acc1 = BankAccount("Олена Петренко", 1000)
acc2 = BankAccount("Іван Коваленко", 500)

print(acc1)
print(acc2)
print()

# Операції
acc1.deposit(500)
acc1.withdraw(200)
acc1.transfer(acc2, 300)

print("\n" + "=" * 60)
print("ПІДСУМКОВІ БАЛАНСИ:")
print(acc1)
print(acc2)

acc1.show_transactions()
acc2.show_transactions()

```

Практичний приклад: клас Library

```

class Book:
    """Клас для представлення книги"""

    def __init__(self, title, author, year, isbn):
        self.title = title
        self.author = author
        self.year = year

```

```

self.isbn = isbn
self.is_available = True
self.borrowed_by = None
self.borrowed_date = None

def __str__(self):
    status = "✓ доступна" if self.is_available else f"✗ видана {self.borrowed_by}"
    return f"{'{self.title}' '{self.author}' ({self.year}) - {status}"

class Library:
    """Клас для представлення бібліотеки"""

    def __init__(self, name):
        self.name = name
        self.books = []
        self.readers = {}

    def add_book(self, book):
        """Додає книгу до бібліотеки"""
        self.books.append(book)
        print(f"✓ Книгу '{book.title}' додано до бібліотеки")

    def find_book(self, title):
        """Шукає книгу за назвою"""
        results = [book for book in self.books if title.lower() in book.title.lower()]
        return results

    def borrow_book(self, isbn, reader_name):
        """Видає книгу читачу"""
        for book in self.books:
            if book.isbn == isbn:
                if not book.is_available:
                    print(f"✗ Книга '{book.title}' вже видана")
                    return False

                book.is_available = False

```

```

book.borrowed_by = reader_name
from datetime import datetime
book.borrowed_date = datetime.now()

# Запам'ятовуємо читача
if reader_name not in self.readers:
    self.readers[reader_name] = []
self.readers[reader_name].append(book)

print(f"✓ Книгу '{book.title}' видано читачу {reader_name}")
return True

print(f"✗ Книгу з ISBN {isbn} не знайдено")
return False

def return_book(self, isbn):
    """Повертає книгу"""
    for book in self.books:
        if book.isbn == isbn:
            if book.is_available:
                print(f"✗ Книга '{book.title}' не була видана")
                return False

            book.is_available = True
            book.borrowed_by = None
            book.borrowed_date = None
            print(f"✓ Книгу '{book.title}' повернено")
            return True

    print(f"✗ Книгу з ISBN {isbn} не знайдено")
    return False

def list_available_books(self):
    """Показує доступні книги"""
    available = [book for book in self.books if book.is_available]
    print(f"\n📖 Доступні книги в бібліотеці '{self.name}':")

```

```

if not available:
    print(" Немає доступних книг")
else:
    for book in available:
        print(f" {book}")

def list_borrowed_books(self):
    """Показує видані книги"""
    borrowed = [book for book in self.books if not book.is_available]
    print(f"\n📖 Видані книги:")
    if not borrowed:
        print(" Немає виданих книг")
    else:
        for book in borrowed:
            print(f" {book}")

def reader_info(self, reader_name):
    """Інформація про читача"""
    if reader_name in self.readers:
        books = self.readers[reader_name]
        print(f"\n👤 Читач {reader_name} взяв {len(books)} книг:")
        for book in books:
            print(f" {book}")
    else:
        print(f"❌ Читач {reader_name} не знайдений")

def __str__(self):
    return f"Бібліотека '{self.name}': {len(self.books)} книг,
{len(self.readers)} читачів"

# Демонстрація
print("📖 СИСТЕМА БІБЛІОТЕКИ")
print("=" * 60)

# Створюємо бібліотеку
library = Library("Міська бібліотека")

```

Додаємо книги

```
library.add_book(Book("Майстер і Маргарита", "М. Булгаков", 1967, "978-5-699-12345-6"))  
library.add_book(Book("Злочин і кара", "Ф. Достоевський", 1866, "978-5-699-12345-7"))  
library.add_book(Book("Війна і мир", "Л. Толстой", 1869, "978-5-699-12345-8"))  
library.add_book(Book("Енеїда", "І. Котляревський", 1798, "978-5-699-12345-9"))
```

```
print("\n" + "=" * 60)
```

Видаємо книги

```
library.borrow_book("978-5-699-12345-6", "Олена Петренко")  
library.borrow_book("978-5-699-12345-7", "Іван Коваленко")
```

```
library.list_available_books()  
library.list_borrowed_books()
```

Повертаємо книгу

```
library.return_book("978-5-699-12345-6")
```

```
print("\n" + "=" * 60)  
library.list_available_books()  
library.reader_info("Іван Коваленко")
```

```
print(f"\n{library}")
```

Цей приклад демонструє:

- Створення класів з конструкторами `__init__`
- Атрибути об'єктів (дані)
- Методи для роботи з об'єктами
- Взаємодію між об'єктами різних класів
- Інкапсуляцію (дані та методи роботи з ними в одному місці)

У наступних пунктах ми розглянемо атрибути класу та екземпляра, різні типи методів, а також магічні методи, які роблять наші класи ще потужнішими.

3. Атрибути класу та екземпляра: де зберігати дані?

Коли ми працюємо з класами, важливо розуміти, що дані можна зберігати на двох рівнях: на рівні класу (спільні для всіх об'єктів) та на рівні екземпляра (унікальні для кожного об'єкта). Розуміння цієї різниці допомагає писати правильний та ефективний код.

Атрибути екземпляра (instance attributes)

Це атрибути, які належать конкретному об'єкту. Вони визначаються в методі `__init__` через `self`. Кожен об'єкт має свою копію таких атрибутів.

```
class Student:
    """Клас для демонстрації атрибутів екземпляра"""

    def __init__(self, name, age):
        # Атрибути екземпляра
        self.name = name
        self.age = age
        self.grades = [] # кожен студент має свій список оцінок

# Створюємо двох студентів
student1 = Student('Олена', 19)
student2 = Student('Іван', 20)

# Додаємо оцінки різним студентам
student1.grades.append(95)
student1.grades.append(90)
student2.grades.append(85)

print(f"{student1.name}: {student1.grades}") # [95, 90]
print(f"{student2.name}: {student2.grades}") # [85]

# Кожен об'єкт має власний незалежний список
```

Атрибути класу (class attributes)

Це атрибути, які належать самому класу. Всі об'єкти класу мають доступ до них, і вони спільні для всіх об'єктів. Визначаються поза методами, безпосередньо в тілі класу.


```

class Student:
    # Атрибути класу (спільні для всіх студентів)
    school = "НТУУ 'КПІ'"
    total_students = 0 # лічильник створених студентів

    def __init__(self, name, age):
        # Атрибути екземпляра
        self.name = name
        self.age = age
        self.grades = []

        # Збільшимо лічильник при створенні нового студента
        Student.total_students += 1

    def info(self):
        return f"{self.name} (вік {self.age}) навчається в {self.school}"

# Створюємо студентів
student1 = Student('Олена', 19)
student2 = Student('Іван', 20)
student3 = Student('Марія', 18)

print(f"Школа: {Student.school}")
print(f"Всього студентів: {Student.total_students}")

# Доступ через об'єкти теж працює
print(f"student1.school: {student1.school}")
print(f"student2.school: {student2.school}")

# Зміна атрибута класу через клас
Student.school = "КНУ ім. Шевченка"
print(f"Після зміни: {student1.school}")

# АЛЕ! Якщо змінити через об'єкт - створиться новий атрибут
екземпляра
student1.school = "Локальна школа" # створює новий атрибут екземпляра

```

```
print(f"student1.school: {student1.school}")
print(f"student2.school: {student2.school} (залишився атрибут класу)")
print(f"Student.school: {Student.school}")
```

Коли використовувати атрибути класу?

Атрибути класу ідеально підходять для:

- Констант, спільних для всіх об'єктів
- Лічильників об'єктів
- Налаштувань за замовчуванням
- Спільних ресурсів (наприклад, пул з'єднань з базою даних)

Практичний приклад: система обліку співробітників

```
from datetime import datetime
```

```
class Employee:
```

```
    """Клас для представлення співробітника з використанням атрибутів
    класу"""
```

```
    # Атрибути класу (спільні для всіх)
```

```
    company_name = "Технології майбутнього"
```

```
    min_salary = 15000
```

```
    max_salary = 50000
```

```
    total_employees = 0
```

```
    all_employees = [] # список всіх співробітників
```

```
    def __init__(self, name, position, salary):
```

```
        # Атрибути екземпляра
```

```
        self.name = name
```

```
        self.position = position
```

```
        self.salary = self._validate_salary(salary)
```

```
        self.employee_id = Employee.generate_id()
```

```
        self.hire_date = datetime.now()
```

```
    # Оновлюємо атрибути класу
```

```
    Employee.total_employees += 1
```

```
    Employee.all_employees.append(self)
```

```

def _validate_salary(self, salary):
    """Приватний метод для перевірки зарплати"""
    if salary < Employee.min_salary:
        print(f"Зарплата {salary} менша за мінімальну. Встановлено мінімум {Employee.min_salary}")
        return Employee.min_salary
    elif salary > Employee.max_salary:
        print(f"Зарплата {salary} більша за максимальну. Встановлено максимум {Employee.max_salary}")
        return Employee.max_salary
    return salary

@classmethod
def generate_id(cls):
    """Генерує унікальний ID для співробітника"""
    return f"EMP_{len(cls.all_employees) + 1:04d}"

@classmethod
def update_min_salary(cls, new_min):
    """Оновлює мінімальну зарплату для всіх"""
    if new_min > cls.max_salary:
        print(f"Мінімум не може перевищувати максимум ({cls.max_salary})")
        return
    cls.min_salary = new_min
    print(f"✓ Мінімальну зарплату оновлено до {new_min}")

@classmethod
def get_company_stats(cls):
    """Повертає статистику по компанії"""
    return {
        'company': cls.company_name,
        'total_employees': cls.total_employees,
        'avg_salary': sum(e.salary for e in cls.all_employees) /
cls.total_employees if cls.total_employees else 0,
        'min_salary': cls.min_salary,
        'max_salary': cls.max_salary
    }

```

```
}
```

```
def give_raise(self, amount):  
    """Підвищує зарплату співробітнику"""  
    new_salary = self.salary + amount  
    if new_salary > Employee.max_salary:  
        print(f"✗ Підвищення перевищує максимум  
({Employee.max_salary})")  
        return False  
    self.salary = new_salary  
    print(f"✓ {self.name} отримав підвищення! Нова зарплата:  
{self.salary}")  
    return True
```

```
def __str__(self):  
    return f"{self.employee_id}: {self.name} ({self.position}) - {self.salary}  
грн"
```

Демонстрація

```
print("👋 СИСТЕМА ОБЛІКУ СПІВРОБІТНИКІВ")  
print("=" * 70)
```

Створюємо співробітників

```
emp1 = Employee("Олена Петренко", "Програміст", 35000)  
emp2 = Employee("Іван Коваленко", "Тестувальник", 28000)  
emp3 = Employee("Марія Шевченко", "Менеджер", 45000)
```

```
print("\n📁 Список співробітників:")  
for emp in Employee.all_employees:  
    print(f" {emp}")
```

```
print(f"\n📊 Статистика компанії:")  
stats = Employee.get_company_stats()  
for key, value in stats.items():  
    print(f" {key}: {value}")
```

Змінюємо мінімальну зарплату (впливає на всіх майбутніх)

```

print("\n" + "=" * 70)
Employee.update_min_salary(20000)

# Створюємо нового співробітника з зарплатою нижче мінімуму
emp4 = Employee("Петро Бондаренко", "Стажер", 18000)
print(f" {emp4}")

# Підвищення
print("\n" + "=" * 70)
emp1.give_raise(5000)

print(f"\n📊 Оновлена статистика:")
stats = Employee.get_company_stats()
for key, value in stats.items():
    print(f" {key}: {value}")

```

Важливе зауваження про змінні типи як атрибути класу

Будьте обережні з атрибутами класу, які є змінними типами (списки, словники). Вони спільні для всіх об'єктів, і зміни в одному об'єкті вплинуть на інші!

```

class ProblematicClass:
    # ПОГАНО! Список як атрибут класу
    shared_list = []

    def __init__(self, value):
        self.value = value
        self.shared_list.append(value) # всі об'єкти змінюють один список!

# Створюємо об'єкти
obj1 = ProblematicClass(1)
obj2 = ProblematicClass(2)
obj3 = ProblematicClass(3)

print(f"obj1.shared_list: {obj1.shared_list}") # [1, 2, 3]
print(f"obj2.shared_list: {obj2.shared_list}") # [1, 2, 3]
print(f"obj3.shared_list: {obj3.shared_list}") # [1, 2, 3]

```

Якщо ми хотіли, щоб кожен об'єкт мав свій список, це помилка!

ПРАВИЛЬНО

```
class GoodClass:
    def __init__(self, value):
        self.value = value
        self.my_list = [] # атрибут екземпляра
        self.my_list.append(value)

obj1 = GoodClass(1)
obj2 = GoodClass(2)

print(f"obj1.my_list: {obj1.my_list}") # [1]
print(f"obj2.my_list: {obj2.my_list}") # [2]
```

4. Методи класу, статичні методи та магічні методи

У Python є різні типи методів, кожен з яких має своє призначення. Розуміння цих типів допомагає писати більш чистий та організований код.

Типи методів

1. **Методи екземпляра** – звичайні методи, які ми вже використовували. Вони отримують self як перший параметр і працюють з конкретним об'єктом.

2. **Методи класу** (@classmethod) – отримують cls як перший параметр і працюють з класом, а не з екземпляром.

3. **Статичні методи** (@staticmethod) – не отримують ні self, ні cls. Це звичайні функції всередині класу.

```
class DemoClass:
    """Демонстрація різних типів методів"""

    class_attr = "атрибут класу"

    def __init__(self, value):
        self.instance_attr = value
```

1. Звичайний метод екземпляра

```
def instance_method(self):
    """Працює з екземпляром, має доступ до self"""
    return f"Екземпляр: {self.instance_attr}, клас: {self.class_attr}"
```

2. Метод класу

```
@classmethod
def class_method(cls):
    """Працює з класом, має доступ до cls"""
    return f"Метод класу: {cls.class_attr}"
```

3. Статичний метод

```
@staticmethod
def static_method(x, y):
    """Не має доступу ні до класу, ні до екземпляра"""
    return f"Статичний метод: {x} + {y} = {x + y}"
```

Демонстрація

```
obj = DemoClass("значення")
```

Виклик методів екземпляра

```
print(obj.instance_method())
```

Виклик методу класу (можна через клас або через об'єкт)

```
print(DemoClass.class_method())
```

```
print(obj.class_method())
```

Виклик статичного методу (можна через клас або через об'єкт)

```
print(DemoClass.static_method(10, 20))
```

```
print(obj.static_method(5, 7))
```

Коли використовувати різні типи методів?

- **Методи екземпляра** – коли метод потребує доступу до даних конкретного об'єкта
- **Методи класу** – коли метод потребує доступу до атрибутів класу або створює об'єкти класу (фабричні методи)
- **Статичні методи** – коли метод логічно належить до класу, але не потребує доступу ні до класу, ні до екземпляра

Практичний приклад: клас Date з різними типами методів

```
class Date:
```

```
    """Клас для роботи з датами, демонструє різні типи методів"""
```

```
    # Атрибути класу
```

```
    months = {
```

```
        1: "січня", 2: "лютого", 3: "березня", 4: "квітня",
```

```
        5: "травня", 6: "червня", 7: "липня", 8: "серпня",
```

```
        9: "вересня", 10: "жовтня", 11: "листопада", 12: "грудня"
```

```
    }
```

```
    def __init__(self, day, month, year):
```

```
        self.day = day
```

```
        self.month = month
```

```
        self.year = year
```

```
    # Звичайний метод екземпляра
```

```
    def format_uk(self):
```

```
        """Повертає дату в українському форматі"""
```

```
        month_name = self.months[self.month]
```

```
        return f"{self.day} {month_name} {self.year} року"
```

```
    def format_iso(self):
```

```
        """Повертає дату в міжнародному форматі (ISO)"""
```

```
        return f"{self.year:04d}-{self.month:02d}-{self.day:02d}"
```

```
    def days_in_month(self):
```

```
        """Повертає кількість днів у місяці"""
```

```
        return Date._days_in_month(self.month, self.year)
```

```
    # Статичний метод – логічно належить до класу, але не потребує self/cls
```

```
    @staticmethod
```

```
    def is_leap_year(year):
```

```
        """Перевіряє, чи є рік високосним"""
```

```
        return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
```



```

@staticmethod
def _days_in_month(month, year):
    """Внутрішній метод для визначення днів у місяці"""
    if month in [4, 6, 9, 11]:
        return 30
    elif month == 2:
        return 29 if Date.is_leap_year(year) else 28
    else:
        return 31

# Метод класу – фабричний метод для створення об'єктів
@classmethod
def from_string(cls, date_str, format='iso'):
    """Створює об'єкт Date з рядка"""
    if format == 'iso':
        # Очікується формат "2023-12-25"
        parts = date_str.split('-')
        if len(parts) == 3:
            year, month, day = map(int, parts)
            return cls(day, month, year)

    elif format == 'uk':
        # Очікується формат "25.12.2023"
        parts = date_str.split('.')
        if len(parts) == 3:
            day, month, year = map(int, parts)
            return cls(day, month, year)

    raise ValueError(f"Неправильний формат дати: {date_str}")

@classmethod
def today(cls):
    """Повертає об'єкт Date з поточною датою"""
    from datetime import date
    today = date.today()
    return cls(today.day, today.month, today.year)

```

Демонстрація

```
print("📅 РОБОТА З ДАТАМИ")
print("=" * 70)
```

Створення об'єктів різними способами

```
d1 = Date(25, 12, 2023)
d2 = Date.from_string("2023-12-25", 'iso')
d3 = Date.from_string("25.12.2023", 'uk')
d4 = Date.today()
```

```
print("Різні способи створення дат:")
print(f" d1 (пряме): {d1.format_uk()}")
print(f" d2 (з ISO): {d2.format_uk()}")
print(f" d3 (з укр): {d3.format_uk()}")
print(f" d4 (сьогодні): {d4.format_uk()}")
```

```
print(f"\n📅 Статичні методи:")
print(f" 2024 високосний? {Date.is_leap_year(2024)}")
print(f" 2023 високосний? {Date.is_leap_year(2023)}")
```

```
print(f"\n📅 Кількість днів у місяцях:")
for month in range(1, 13):
    days = Date._days_in_month(month, 2024)
    print(f" {Date.months[month]}: {days} днів")
```

Магічні методи (dunder methods)

Магічні методи – це спеціальні методи з подвійним підкресленням (`__method__`), які дозволяють нашим класам поводитися як вбудовані типи Python. Вони викликаються автоматично при певних операціях.

```
class Book:
    """Клас книги з магічними методами"""

    def __init__(self, title, author, pages, price):
        self.title = title
        self.author = author
```

```

self.pages = pages
self.price = price
self._id = id(self)

# __str__ – для користувачів (красиве виведення)
def __str__(self):
    return f'{{self.title}} – {{self.author}}'

# __repr__ – для розробників (однозначне представлення)
def __repr__(self):
    return f"Book('{{self.title}}', '{{self.author}}', {{self.pages}}, {{self.price}})"

# __len__ – довжина (кількість сторінок)
def __len__(self):
    return self.pages

# __eq__ – порівняння на рівність
def __eq__(self, other):
    if not isinstance(other, Book):
        return False
    return self.title == other.title and self.author == other.author

# __lt__ – менше (для сортування за ціною)
def __lt__(self, other):
    if not isinstance(other, Book):
        return NotImplemented
    return self.price < other.price

# __add__ – додавання (наприклад, об'єднання книг)
def __add__(self, other):
    if isinstance(other, Book):
        return Book(
            f'{{self.title}} + {{other.title}}',
            f'{{self.author}} & {{other.author}}',
            self.pages + other.pages,
            self.price + other.price
        )

```

```

    return NotImplemented

# __contains__ – перевірка належності
def __contains__(self, item):
    return item.lower() in self.title.lower() or item.lower() in self.author.lower()

# __call__ – об'єкт можна викликати як функцію
def __call__(self, discount):
    """Застосовує знижку до ціни"""
    self.price *= (1 - discount / 100)
    return f"Нова ціна зі знижкою {discount}%: {self.price:.2f}"

print("🪄 МАГІЧНІ МЕТОДИ")
print("=" * 70)

# Створюємо книги
book1 = Book("Майстер і Маргарита", "М. Булгаков", 480, 350)
book2 = Book("Злочин і кара", "Ф. Достоевський", 670, 420)
book3 = Book("Майстер і Маргарита", "М. Булгаков", 480, 350)

# __str__ та __repr__
print("__str__:", str(book1))
print("__repr__:", repr(book1))

# __len__
print(f"Кількість сторінок: {len(book1)}")

# __eq__
print(f"book1 == book2: {book1 == book2}")
print(f"book1 == book3: {book1 == book3}")

# __lt__ (для сортування)
books = [book1, book2]
books.sort()
print("Відсортовано за ціною:")
for book in books:
    print(f"{book} – {book.price} грн")

```

```
# __add__
book4 = book1 + book2
print(f"Об'єднання книг: {book4}")

# __contains__
print(f"‘Майстер’ в book1? {'Майстер' in book1}")
print(f"‘Булгаков’ в book1? {'Булгаков' in book1}")
print(f"‘Пушкін’ в book1? {'Пушкін' in book1}")

# __call__
print(book1(10)) # застосовуємо знижку 10%
```

Найважливіші магічні методи

Метод	Опис	Викликається
<code>__init__</code>	Ініціалізатор	При створенні об'єкта
<code>__str__</code>	Рядкове представлення	<code>str(obj)</code> , <code>print(obj)</code>
<code>__repr__</code>	Офіційне представлення	<code>repr(obj)</code> , в інтерактивному режимі
<code>__len__</code>	Довжина	<code>len(obj)</code>
<code>__contains__</code>	Перевірка належності	<code>item in obj</code>
<code>__getitem__</code>	Доступ за індексом	<code>obj[key]</code>
<code>__setitem__</code>	Встановлення за індексом	<code>obj[key] = value</code>
<code>__delitem__</code>	Видалення за індексом	<code>del obj[key]</code>
<code>__call__</code>	Виклик об'єкта	<code>obj()</code>
<code>__eq__</code>	Рівність	<code>obj1 == obj2</code>
<code>__lt__</code>	Менше	<code>obj1 < obj2</code>

Метод	Опис	Викликається
<code>__add__</code>	Додавання	<code>obj1 + obj2</code>
<code>__enter__</code> / <code>__exit__</code>	Контекстний менеджер	<code>with obj:</code>

Практичний приклад: клас `Vector` з магічними методами

```
import math
```

```
class Vector:
```

```
    """Клас для роботи з векторами у 2D просторі"""
```

```
    def __init__(self, x, y):
```

```
        self.x = x
```

```
        self.y = y
```

```
    def __str__(self):
```

```
        return f"Vector({self.x}, {self.y})"
```

```
    def __repr__(self):
```

```
        return f"Vector({self.x}, {self.y})"
```

```
    def __abs__(self):
```

```
        """Довжина вектора"""
```

```
        return math.sqrt(self.x**2 + self.y**2)
```

```
    def __bool__(self):
```

```
        """Вектор вважається істинним, якщо його довжина > 0"""
```

```
        return abs(self) > 0
```

```
    def __add__(self, other):
```

```
        """Додавання векторів"""
```

```
        if isinstance(other, Vector):
```

```
            return Vector(self.x + other.x, self.y + other.y)
```

```
        return NotImplemented
```

```
    def __sub__(self, other):
```

```

        """Віднімання векторів"""
        if isinstance(other, Vector):
            return Vector(self.x - other.x, self.y - other.y)
        return NotImplemented

    def __mul__(self, scalar):
        """Множення на скаляр"""
        if isinstance(scalar, (int, float)):
            return Vector(self.x * scalar, self.y * scalar)
        return NotImplemented

    def __rmul__(self, scalar):
        """Множення на скаляр зліва (скаляр * вектор)"""
        return self.__mul__(scalar)

    def __matmul__(self, other):
        """Скалярний добуток (оператор @)"""
        if isinstance(other, Vector):
            return self.x * other.x + self.y * other.y
        return NotImplemented

    def __eq__(self, other):
        """Порівняння векторів"""
        if not isinstance(other, Vector):
            return False
        return self.x == other.x and self.y == other.y

    def __neg__(self):
        """Унарний мінус"""
        return Vector(-self.x, -self.y)

    def __getitem__(self, index):
        """Доступ за індексом (v[0] -> x, v[1] -> y)"""
        if index == 0:
            return self.x
        elif index == 1:
            return self.y

```

```

        raise IndexError("Vector index out of range")

    def __setitem__(self, index, value):
        """Встановлення за індексом"""
        if index == 0:
            self.x = value
        elif index == 1:
            self.y = value
        else:
            raise IndexError("Vector index out of range")

# Демонстрація
print("📐 ВЕКТОРНИЙ КАЛЬКУЛЯТОР")
print("=" * 70)

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

print(f"v1 = {v1}")
print(f"v2 = {v2}")

print(f"\nДовжина v1: {abs(v1):.2f}")
print(f"v1 + v2 = {v1 + v2}")
print(f"v1 - v2 = {v1 - v2}")
print(f"v1 * 2 = {v1 * 2}")
print(f"3 * v1 = {3 * v1}")
print(f"Скалярний добуток v1 @ v2 = {v1 @ v2}")
print(f"v1 == Vector(3, 4)? {v1 == Vector(3, 4)}")
print(f"v1 == v2? {v1 == v2}")
print(f"-v1 = {-v1}")

print(f"\nДоступ за індексами: v1[0] = {v1[0]}, v1[1] = {v1[1]}")
v1[0] = 10
print(f"Після v1[0] = 10: {v1}")

print(f"Чи є вектор істинним? {bool(v1)}")
print(f"Чи є нульовий вектор істинним? {bool(Vector(0, 0))}")

```


Підсумки

- **Атрибути класу** спільні для всіх об'єктів, **атрибути екземпляра** унікальні для кожного
- **Методи екземпляра** працюють з конкретним об'єктом (self)
- **Методи класу** (@classmethod) працюють з класом (cls) – ідеально для фабрик
- **Статичні методи** (@staticmethod) – звичайні функції в просторі імен класу
- **Магічні методи** дозволяють інтегрувати класи з синтаксисом Python

У наступній лекції ми розглянемо наслідування та поліморфізм – ключові концепції ООП, які дозволяють створювати ієрархії класів і повторно використовувати код.

Питання для обговорення

1. Поясніть різницю між класом та об'єктом. Наведіть аналогію з реального світу (наприклад, форма для випічки та печиво). Чому важливо розрізняти ці поняття?
2. Що таке конструктор класу і як він позначається в Python? Чому метод `__init__` не створює об'єкт, а лише ініціалізує його? Де і коли створюється сам об'єкт?
3. Поясніть призначення параметра `self` у методах класу. Чи можна назвати його інакше? Що станеться, якщо забути вказати `self` як перший параметр методу?
4. Проаналізуйте код:

```
class Student:
    school = "НТУУ КПІ"

    def __init__(self, name, age):
        self.name = name
        self.age = age

s1 = Student("Олена", 19)
s2 = Student("Іван", 20)
```

Поясніть, які атрибути є атрибутами класу, а які – атрибутами екземпляра. Як змінити назву школи для всіх студентів? Як змінити назву школи тільки для одного студента?

5. Напишіть клас `BankAccount` з атрибутами `owner` (ім'я власника) та `balance` (баланс). Додайте методи `deposit(amount)` та `withdraw(amount)`. Метод `withdraw` не повинен дозволяти зняти більше, ніж є на рахунку.

6. Чому методи класу отримують параметр `cls`, а статичні методи не отримують ні `self`, ні `cls`? Наведіть приклад, коли доцільно використовувати метод класу, а коли – статичний метод.

7. Проаналізуйте код:

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

    def __str__(self):
        return f"Значення: {self.value}"

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

Чим відрізняються методи `__str__` та `__repr__`? В яких ситуаціях викликається кожен з них?

8. Що таке інкапсуляція? Як в Python реалізується приховування внутрішніх даних? Поясніть різницю між захищеними (`_attr`) та приватними (`__attr`) атрибутами.

9. Напишіть клас `Rectangle` з атрибутами `width` та `height`. Додайте обчислювану властивість `area`, яка повертає площу прямокутника. Зробіть так, щоб при зміні ширини або висоти площа перераховувалася автоматично.

10. Поясніть поняття "магічних методів" (dunder methods). Наведіть приклади магічних методів для арифметичних операцій (`__add__`, `__sub__`) та для порівняння (`__eq__`, `__lt__`).

ЛЕКЦІЯ 15

Наслідування та поліморфізм

План лекції

1. Концепція наслідування: створення ієрархій класів.

Пояснимо, що таке наслідування, навіщо воно потрібне, як воно допомагає повторно використовувати код. Розберемо базовий (батьківський) та похідний (дочірній) класи, синтаксис наслідування в Python. Покажемо, як дочірній клас отримує всі атрибути та методи батьківського.

2. Перевизначення методів та розширення функціональності.

Навчимося змінювати поведінку успадкованих методів у дочірніх класах. Розберемо, як викликати методи батьківського класу за допомогою `super()`. Покажемо, як можна як повністю замінити метод, так і розширити його, додавши нову функціональність.

3. Множинне наслідування та MRO.

Розглянемо особливість Python – можливість успадковувати від кількох класів. Пояснимо, як Python вирішує конфлікти імен за допомогою MRO (Method Resolution Order). Покажемо, коли множинне наслідування корисне, а коли краще його уникати.

4. Поліморфізм та абстрактні класи.

Розберемо концепцію поліморфізму – одну з ключових в ООП. Пояснимо "качину типізацію" (duck typing) в Python. Pozнайомимося з абстрактними класами та методами, які дозволяють створювати інтерфейси та гарантувати, що дочірні класи реалізують певні методи.

1. Концепція наслідування: створення ієрархій класів

У реальному світі ми постійно стикаємося з ієрархіями. Наприклад, "собака" є окремим випадком "тварини", "легковий автомобіль" є окремим випадком "транспортного засобу", "викладач" є окремим випадком "людини". У програмуванні ці відношення "є" (is-a) моделюються за допомогою наслідування.

Що таке наслідування?

Наслідування – це механізм ООП, який дозволяє створити новий клас на основі вже існуючого. Новий клас (дочірній, похідний) успадковує всі атрибути та методи батьківського (базового) класу і може додавати свої власні або змінювати успадковані.

Навіщо потрібне наслідування?

1. **Повторне використання коду** – не потрібно писати той самий код для різних класів
2. **Структурування коду** – створення логічних ієрархій
3. **Поліморфізм** – можливість працювати з різними об'єктами через спільний інтерфейс
4. **Розширення функціональності** – легко додавати нові можливості до існуючих класів

Базовий синтаксис наслідування

```
class БатьківськийКлас:  
    # тіло батьківського класу  
    pass  
  
class ДочірнійКлас(БатьківськийКлас):  
    # тіло дочірнього класу  
    pass
```

Простий приклад

```
class Animal:  
    """Базовий клас для всіх тварин"""  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def speak(self):  
        """Загальний метод для звуків тварини"""  
        return f"{self.name} видає звук"  
  
    def info(self):  
        return f"{self.name}, {self.age} років"  
  
class Dog(Animal):  
    """Клас собаки, успадкований від Animal"""  
    pass # поки що нічого не додаємо
```

```
class Cat(Animal):
    """Клас kota, успадкований від Animal"""
    pass
```

Створюємо об'єкти

```
dog = Dog("Бобік", 3)
cat = Cat("Мурка", 2)
```

Викликаємо успадковані методи

```
print(dog.info())
print(dog.speak())
print(cat.info())
print(cat.speak())
```

Навіть без додавання коду в дочірні класи, вони вже мають всі методи та атрибути батьківського класу!

Перевірка належності до класу

```
print(f"dog є екземпляром Dog? {isinstance(dog, Dog)}")
print(f"dog є екземпляром Animal? {isinstance(dog, Animal)}")
print(f"dog є екземпляром Cat? {isinstance(dog, Cat)}")
```

```
print(f"Dog є підкласом Animal? {issubclass(Dog, Animal)}")
print(f"Animal є підкласом Dog? {issubclass(Animal, Dog)}")
```

Додавання нових атрибутів та методів

Дочірні класи можуть мати власні атрибути та методи, яких немає в батьківському класі.

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        return f"{self.name}, {self.age} років"

class Dog(Animal):
```

```

def __init__(self, name, age, breed):
    # Викликаємо конструктор батьківського класу
    super().__init__(name, age)
    self.breed = breed # новий атрибут

def wag_tail(self): # новий метод
    return f"{self.name} виляє хвостом"

class Cat(Animal):
    def __init__(self, name, age, color):
        super().__init__(name, age)
        self.color = color # новий атрибут

    def purr(self): # новий метод
        return f"{self.name} муркоче"

# Створюємо об'єкти
dog = Dog("Бобік", 3, "лабрадор")
cat = Cat("Мурка", 2, "руда")

print(dog.info())
print(f"Порода: {dog.breed}")
print(dog.wag_tail())

print(cat.info())
print(f"Колір: {cat.color}")
print(cat.purr())

```

Чому super()?

super() дозволяє викликати методи батьківського класу. Це важливо, тому що:

- Ми не хочемо дублювати код ініціалізації
- Батьківський клас може робити важливі речі у своєму конструкторі
- При множинному наслідуванні super() правильно вирішує порядок викликів

викликів

Практичний приклад: ієрархія транспортних засобів

```

class Vehicle:
    """Базовий клас для всіх транспортних засобів"""

    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
        self.speed = 0
        self.is_started = False

    def start(self):
        self.is_started = True
        return f"{self.brand} {self.model} запущено"

    def stop(self):
        self.is_started = False
        self.speed = 0
        return f"{self.brand} {self.model} зупинено"

    def accelerate(self, increment):
        if not self.is_started:
            return f"Спочатку запустіть двигун"
        self.speed += increment
        return f"Швидкість: {self.speed} км/год"

    def brake(self, decrement):
        self.speed = max(0, self.speed - decrement)
        return f"Швидкість: {self.speed} км/год"

    def info(self):
        return f"{self.brand} {self.model} ({self.year})"

class Car(Vehicle):
    """Клас автомобіля"""

    def __init__(self, brand, model, year, doors, fuel_type):
        super().__init__(brand, model, year)

```

```

self.doors = doors
self.fuel_type = fuel_type
self.gear = 0

def shift_gear(self, gear):
    self.gear = gear
    return f'Передача {gear}'

def info(self): # перевизначення методу
    return f'Автомобіль: {super().info()}, {self.doors} дверей,
{self.fuel_type}'

class Motorcycle(Vehicle):
    """Клас мотоцикла"""

    def __init__(self, brand, model, year, engine_size):
        super().__init__(brand, model, year)
        self.engine_size = engine_size
        self.has_sidecar = False

    def wheelie(self):
        if self.speed > 50:
            return f'{self.brand} {self.model} їде на задньому колесі! 🏍️'
        return f'Швидкість занадто мала для віллі'

    def info(self):
        return f'Мотоцикл: {super().info()}, {self.engine_size}cc'

class Truck(Vehicle):
    """Клас вантажівки"""

    def __init__(self, brand, model, year, capacity):
        super().__init__(brand, model, year)
        self.capacity = capacity # вантажопідйомність в тоннах
        self.current_load = 0

    def load(self, weight):

```



```

    if self.current_load + weight > self.capacity:
        return f"Перевантаження! Максимум {self.capacity}т"
    self.current_load += weight
    return f"Завантажено {weight}т. Поточне завантаження: {self.current_load}т"

```

```

def unload(self, weight):
    self.current_load = max(0, self.current_load - weight)
    return f"Розвантажено {weight}т. Залишок: {self.current_load}т"

```

```

def info(self):
    return f"Вантажівка: {super().info()}, {self.capacity}т"

```

Демонстрація

```

print("🚗 ТРАНСПОРТНІ ЗАСОБИ")
print("=" * 70)

```

Створюємо різні транспортні засоби

```

car = Car("Toyota", "Camry", 2020, 4, "бензин")
moto = Motorcycle("Honda", "CBR600RR", 2022, 600)
truck = Truck("Volvo", "FH16", 2021, 20)

```

```

print(car.info())
print(car.start())
print(car.accelerate(30))
print(car.shift_gear(2))
print(car.accelerate(50))
print(car.brake(20))
print(car.stop())

```

```

print("\n" + "-" * 40)
print(moto.info())
print(moto.start())
print(moto.accelerate(60))
print(moto.wheelie())
print(moto.stop())

```

```
print("\n" + "-" * 40)
print(truck.info())
print(truck.start())
print(truck.load(5))
print(truck.load(10))
print(truck.load(10)) # перевантаження!
print(truck.accelerate(40))
```

2. Перевизначення методів та розширення функціональності

Одна з найпотужніших можливостей наслідування — це здатність змінювати поведінку успадкованих методів у дочірніх класах. Це називається **перевизначенням (overriding)** методів.

Повне перевизначення методу

Іноді метод батьківського класу зовсім не підходить для дочірнього, і ми хочемо повністю замінити його поведінку.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Деякий звук тварини"

class Dog(Animal):
    def speak(self): # повністю перевизначаємо метод
        return "Гав-гав!"

class Cat(Animal):
    def speak(self): # повністю перевизначаємо метод
        return "Мяу!"

class Cow(Animal):
    def speak(self): # повністю перевизначаємо метод
        return "Му-у-у!"
```

Демонстрація

```
animals = [Dog("Бобік"), Cat("Мурка"), Cow("Бурьонка")]
```

```
for animal in animals:
```

```
    print(f'{animal.name}: {animal.speak()}')
```

Розширення методу (виклик батьківського методу)

Часто ми хочемо не замінити метод повністю, а додати до нього нову функціональність, зберігши те, що робить батьківський метод. Для цього використовуємо `super()`.

```
class Employee:
```

```
    """Базовий клас співробітника"""
```

```
    def __init__(self, name, salary):
```

```
        self.name = name
```

```
        self.salary = salary
```

```
        print(f"✓ Співробітника {name} створено")
```

```
    def work(self):
```

```
        return f'{self.name} працює'
```

```
    def get_info(self):
```

```
        return f'Ім'я: {self.name}, Зарплата: {self.salary} грн'
```

```
class Manager(Employee):
```

```
    def __init__(self, name, salary, department):
```

```
        # Викликаємо конструктор батьківського класу
```

```
        super().__init__(name, salary)
```

```
        # Додаємо нові атрибути
```

```
        self.department = department
```

```
        self.team = []
```

```
        print(f" Менеджер відділу {department}")
```

```
    def work(self):
```

```
        # Розширюємо метод work
```

```
        base_message = super().work()
```

```
        return f'{base_message} та керує відділом {self.department}'
```

```

def add_to_team(self, employee):
    self.team.append(employee)
    return f"{employee.name} додано до команди {self.name}"

def get_info(self):
    # Розширюємо метод get_info
    base_info = super().get_info()
    return f"{base_info}, Відділ: {self.department}, Команда: {len(self.team)} осіб"

```

```

class Developer(Employee):
    def __init__(self, name, salary, programming_language):
        super().__init__(name, salary)
        self.programming_language = programming_language
        print(f"Розробник на {programming_language}")

    def work(self):
        base_message = super().work()
        return f"{base_message} та пише код на {self.programming_language}"

    def get_info(self):
        base_info = super().get_info()
        return f"{base_info}, Мова: {self.programming_language}"

```

```

# Демонстрація
print("👑 ІЄРАРХІЯ СПІВРОБІТНИКІВ")
print("=" * 70)

```

```

# Створюємо співробітників
dev1 = Developer("Олена Петренко", 35000, "Python")
dev2 = Developer("Іван Коваленко", 32000, "JavaScript")
manager = Manager("Марія Шевченко", 45000, "Розробка")

```

```

# Додаємо розробників до команди менеджера
print(f"\n{manager.add_to_team(dev1)}")
print(manager.add_to_team(dev2))

```

Демонструємо роботу

```
print("\n📅 РОБОЧИЙ ДЕНЬ:")
```

```
print(dev1.work())
```

```
print(dev2.work())
```

```
print(manager.work())
```

```
print("\n📋 ІНФОРМАЦІЯ ПРО СПІВРОБІТНИКІВ:")
```

```
print(dev1.get_info())
```

```
print(dev2.get_info())
```

```
print(manager.get_info())
```

Приклад з різними рівнями наслідування

```
class Shape:
```

```
    """Базовий клас для всіх геометричних фігур"""
```

```
    def __init__(self, color="чорний"):
```

```
        self.color = color
```

```
    def area(self):
```

```
        """Метод має бути перевизначений у дочірніх класах"""
```

```
        raise NotImplementedError("Підкласи повинні реалізувати метод  
area()")
```

```
    def perimeter(self):
```

```
        """Метод має бути перевизначений у дочірніх класах"""
```

```
        raise NotImplementedError("Підкласи повинні реалізувати метод  
perimeter()")
```

```
    def info(self):
```

```
        return f"Фігура кольору {self.color}, площа: {self.area():.2f}"
```

```
class Rectangle(Shape):
```

```
    """Прямокутник"""
```

```
    def __init__(self, width, height, color="чорний"):
```

```

    super().__init__(color)
    self.width = width
    self.height = height

def area(self):
    return self.width * self.height

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

def info(self):
    return f"Прямокутник {self.width}x{self.height}, {super().info()}"

class Square(Rectangle):
    """Квадрат (окремий випадок прямокутника)"""

    def __init__(self, side, color="чорний"):
        # Викликаємо конструктор Rectangle з однаковими сторонами
        super().__init__(side, side, color)

    def info(self):
        return f"Квадрат зі стороною {self.width}, {super().info()}"

class Circle(Shape):
    """Коло"""

    def __init__(self, radius, color="чорний"):
        super().__init__(color)
        self.radius = radius

    def area(self):
        import math
        return math.pi * self.radius ** 2

    def perimeter(self):
        import math
        return 2 * math.pi * self.radius

```

```

def info(self):
    return f"Коло радіусом {self.radius}, {super().info()}"

# Демонстрація
print("◆ ГЕОМЕТРИЧНІ ФІГУРИ")
print("=" * 70)

shapes = [
    Rectangle(5, 3, "синій"),
    Square(4, "червоний"),
    Circle(3, "зелений")
]

for shape in shapes:
    print(shape.info())
    print(f" Периметр: {shape.perimeter():.2f}")
    print()

```

Практичний приклад: система платежів

```

from datetime import datetime

class Payment:
    """Базовий клас для різних типів платежів"""

    def __init__(self, amount, currency="UAH"):
        self.amount = amount
        self.currency = currency
        self.timestamp = datetime.now()
        self.status = "pending"

    def process(self):
        """Базовий метод обробки платежу"""
        self.status = "processing"
        return f"Обробка платежу на суму {self.amount} {self.currency}"

    def complete(self):

```

```

        self.status = "completed"
        return f"Платіж завершено"

    def fail(self, reason):
        self.status = "failed"
        return f"Платіж не вдавсь: {reason}"

    def get_info(self):
        return f"Платіж: {self.amount} {self.currency}, статус: {self.status}"

class CreditCardPayment(Payment):
    """Платіж кредитною карткою"""

    def __init__(self, amount, card_number, card_holder, expiry, cvv,
currency="UAH"):
        super().__init__(amount, currency)
        self.card_number = self._mask_card(card_number)
        self.card_holder = card_holder
        self.expiry = expiry
        self.cvv = "****" # ніколи не зберігаємо CVV!

    def _mask_card(self, card_number):
        """Приховує номер картки"""
        return f"**** *{card_number[-4:]}*"

    def process(self):
        # Специфічна логіка для карток
        base_message = super().process()
        # Тут була б комунікація з платіжним шлюзом
        return f"{base_message} через картку {self.card_number}"

class PayPalPayment(Payment):
    """Платіж через PayPal"""

    def __init__(self, amount, email, currency="UAH"):
        super().__init__(amount, currency)
        self.email = email

```



```

def process(self):
    base_message = super().process()
    return f"{base_message} через PayPal ({self.email})"

class CryptoPayment(Payment):
    """Платіж криптовалютою"""

    def __init__(self, amount, wallet_address, crypto_type="BTC",
currency="UAH"):
        super().__init__(amount, currency)
        self.wallet_address = wallet_address
        self.crypto_type = crypto_type
        self.exchange_rate = self._get_exchange_rate()

    def _get_exchange_rate(self):
        # Імітація отримання курсу
        rates = {"BTC": 1500000, "ETH": 90000, "USDT": 38}
        return rates.get(self.crypto_type, 1)

    def process(self):
        base_message = super().process()
        crypto_amount = self.amount / self.exchange_rate
        return (f"{base_message} криптовалютою {self.crypto_type}\n"
            f"        {self.amount} {self.currency} = {crypto_amount:.6f}"
            f"{self.crypto_type}")

class PaymentProcessor:
    """Клас для обробки різних типів платежів"""

    def __init__(self):
        self.payments = []

    def add_payment(self, payment):
        self.payments.append(payment)

    print(f"✓ Додано платіж: {payment.get_info()}")

```

```

def process_all(self):
    print("\n👛 ОБРОБКА ВСІХ ПЛАТЕЖІВ:")
    for payment in self.payments:
        print(f"\n{payment.process()}")
        # Імітуємо успішне завершення
        print(payment.complete())

def get_statistics(self):
    total = sum(p.amount for p in self.payments)
    completed = len([p for p in self.payments if p.status == "completed"])
    return f"Всього платежів: {len(self.payments)}, завершено: {completed},  
сума: {total:.2f} УАН"

# Демонстрація
print("🏠 СИСТЕМА ПЛАТЕЖІВ")
print("=" = 70)

processor = PaymentProcessor()

# Створюємо різні типи платежів
card_payment = CreditCardPayment(
    1500,
    "1234567890123456",
    "OLEH PETRENKO",
    "12/25",
    "123"
)

paypal_payment = PayPalPayment(850, "user@example.com")
crypto_payment = CryptoPayment(50000,
    "1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa", "BTC")

# Додаємо платежі
processor.add_payment(card_payment)
processor.add_payment(paypal_payment)
processor.add_payment(crypto_payment)

```

```
# Обробляємо всі платежі
```

```
processor.process_all()
```

```
print(f"\n{processor.get_statistics()}")
```

Ключові моменти про перевизначення методів:

1. **Повне перевизначення** – коли метод дочірнього класу повністю замінює метод батьківського

2. **Розширення** – коли ми викликаємо батьківський метод через `super()` і додаємо нову функціональність

3. **Збереження інтерфейсу** – навіть при перевизначенні бажано зберігати ту саму сигнатуру методу (ті самі параметри)

4. **NotImplementedError** – в базовому класі можна вказати, що метод має бути перевизначений

У наступному пункті ми розглянемо множинне наслідування – потужний, але потенційно небезпечний механізм Python. < | end_of_thinking | >Дякую! Продовжуємо п'ятнадцяту лекцію. Тепер, коли ми розібрали основи наслідування та перевизначення методів, перейдемо до більш складних тем: множинного наслідування, яке дає ще більше гнучкості, але вимагає обережності, та поліморфізму – однієї з ключових концепцій ООП.

3. Множинне наслідування та MRO

Більшість мов програмування підтримують лише одинарне наслідування (клас може мати лише одного батька). Python підтримує **множинне наслідування** – клас може успадковувати властивості та методи від кількох батьківських класів. Це потужний, але потенційно небезпечний механізм.

Синтаксис множинного наслідування

```
class Base1:  
    pass
```

```
class Base2:  
    pass
```

```
class Derived(Base1, Base2):
```

```
pass
```

Простий приклад

```
class Flyer:
```

```
    """Клас для літаючих істот"""
```

```
    def __init__(self, flight_speed):
        self.flight_speed = flight_speed
```

```
    def fly(self):
        return f"Летить зі швидкістю {self.flight_speed} км/год"
```

```
    def takeoff(self):
        return "Злітає"
```

```
class Swimmer:
```

```
    """Клас для плаваючих істот"""
```

```
    def __init__(self, swim_speed):
        self.swim_speed = swim_speed
```

```
    def swim(self):
        return f"Пливе зі швидкістю {self.swim_speed} км/год"
```

```
    def dive(self):
        return "Пірнає"
```

```
class Duck(Flyer, Swimmer):
```

```
    """Качка – і літає, і плаває"""
```

```
    def __init__(self, name, flight_speed, swim_speed):
        # Потрібно викликати __init__ обох батьківських класів
        Flyer.__init__(self, flight_speed)
        Swimmer.__init__(self, swim_speed)
        self.name = name
```

```
    def quack(self):
```

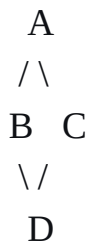
```
return "Кря-кря!"
```

Демонстрація

```
duck = Duck("Каченя", 50, 10)
print(f"{duck.name}:")
print(duck.fly())
print(duck.swim())
print(duck.takeoff())
print(duck.dive())
print(duck.quack())
```

Проблема ромба (Diamond Problem)

Класична проблема множинного наслідування – так званий "ромб", коли клас успадковує від двох класів, які, в свою чергу, успадковують від спільного предка.



Як Python вирішує, який метод викликати, якщо він визначений у кількох батьківських класах?

```
python
class A:
    def method(self):
        return "Метод з A"

class B(A):
    def method(self):
        return "Метод з B"

class C(A):
    def method(self):
        return "Метод з C"

class D(B, C):
    pass
```

Демонстрація

```
d = D()
```

```
print(d.method()) # Що виведе?
```

MRO (Method Resolution Order) – порядок вирішення методів

Python використовує алгоритм СЗ лінеаризації для визначення порядку, в якому шукаються методи. Цей порядок можна подивитися через атрибут `__mro__` або метод `mro()`.

```
print("Порядок вирішення методів для D:")
```

```
for cls in D.__mro__:
```

```
    print(f" {cls.__name__}")
```

Як правильно працювати з множинним наслідуванням?

Ключ до правильного множинного наслідування – використання `super()` замість прямого виклику батьківських класів. `super()` працює з MRO і гарантує, що кожен клас буде викликаний лише один раз.

```
class A:
```

```
    def __init__(self):
```

```
        print("Ініціалізація A")
```

```
        super().__init__()
```

```
class B(A):
```

```
    def __init__(self):
```

```
        print("Ініціалізація B")
```

```
        super().__init__()
```

```
class C(A):
```

```
    def __init__(self):
```

```
        print("Ініціалізація C")
```

```
        super().__init__()
```

```
class D(B, C):
```

```
    def __init__(self):
```

```
        print("Ініціалізація D")
```

```
        super().__init__()
```

```
print("Створення D:")
d = D()
print(f"MRO: {[cls.__name__ for cls in D.__mro__]}")
```

Практичний приклад: система змішаних типів

```
class Animal:
    """Базовий клас для всіх тварин"""

    def __init__(self, name):
        self.name = name
        print(f"Animal.__init__ for {name}")

    def eat(self):
        return f"{self.name} їсть"

    def sleep(self):
        return f"{self.name} спить"

class Mammal(Animal):
    """Ссавці"""

    def __init__(self, name, fur_color):
        super().__init__(name)
        self.fur_color = fur_color
        print(f"Mammal.__init__ for {name}")

    def feed_milk(self):
        return f"{self.name} годує молоком"

class Bird(Animal):
    """Птахи"""

    def __init__(self, name, wing_span):
        super().__init__(name)
        self.wing_span = wing_span
        print(f"Bird.__init__ for {name}")
```

```
def lay_eggs(self):  
    return f"{self.name} відкладає яйця"
```

```
class Bat(Mammal):
```

```
    """Кажани – ссавці, що літають"""
```

```
def __init__(self, name, fur_color, wing_span):  
    super().__init__(name, fur_color)  
    self.wing_span = wing_span  
    print(f"Bat.__init__ for {name}")
```

```
def fly(self):  
    return f"{self.name} літає вночі"
```

```
class Penguin(Bird, Mammal): # Пінгвіни – і птахи, і ссавці? Ні, це для  
прикладу
```

```
    """ПІНГВІН – птах, але з деякими властивостями ссавців (для  
демонстрації)"""
```

```
def __init__(self, name, wing_span, fur_color):  
    # Викликаємо __init__ обох батьків через super()  
    # Завдяки MRO, це спрацює правильно  
    super().__init__(name, wing_span)  
    # Але fur_color потрібно встановити окремо  
    self.fur_color = fur_color  
    print(f"Penguin.__init__ for {name}")
```

```
def swim(self):  
    return f"{self.name} плаває"
```

```
# Демонстрація
```

```
print("🐧 СТВОРЕННЯ ПІНГВІНА:")
```

```
print("=" * 50)
```

```
penguin = Penguin("Пінгвін", 50, "чорно-білий")
```

```
print("\nMRO для Penguin:")
```



```

for i, cls in enumerate(Penguin.__mro__):
    print(f'{i}. {cls.__name__}')

print("\nДії пінгвіна:")
print(penguin.eat())
print(penguin.sleep())
print(penguin.lay_eggs())
print(penguin.swim())

```

Коли використовувати множинне наслідування?

Множинне наслідування потужне, але його варто використовувати обережно. Типові сценарії:

1. **Міксини (Mixins)** – невеликі класи, які додають певну функціональність
2. **Інтерфейси** – класи, які визначають лише методи, але не мають даних
3. **Комбінування незалежних функціональностей**

Приклад з міксинами

```

class LoggerMixin:
    """Міксін для додавання логування"""

    def log(self, message):
        print(f'[LOG] {self.__class__.__name__}: {message}')

    def log_method_call(self, method_name):
        self.log(f'Викликано метод {method_name}')

class TimestampMixin:
    """Міксін для додавання часових міток"""

    def get_timestamp(self):
        from datetime import datetime
        return datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    def log_with_time(self, message):
        print(f'[{self.get_timestamp()}] {message}')

class SerializeMixin:

```

```

"""Міксін для серіалізації в словник"""

def to_dict(self):
    result = {}
    for key, value in self.__dict__.items():
        if not key.startswith('_'): # пропускаємо приватні атрибути
            result[key] = value
    return result

def to_json(self):
    import json
    return json.dumps(self.to_dict(), ensure_ascii=False, indent=2)

class User(LoggerMixin, TimestampMixin, SerializeMixin):
    """Користувач з функціональністю з міксинів"""

    def __init__(self, username, email):
        self.username = username
        self.email = email
        self.log(f"Створено користувача {username}")

    def update_email(self, new_email):
        old_email = self.email
        self.email = new_email
        self.log_method_call("update_email")
        self.log_with_time(f"Email змінено з {old_email} на {new_email}")

class Product(LoggerMixin, SerializeMixin):
    """Товар з функціональністю з міксинів"""

    def __init__(self, name, price):
        self.name = name
        self.price = price
        self.log(f"Створено товар {name}")

    def apply_discount(self, percent):
        old_price = self.price

```

```

self.price *= (1 - percent / 100)
self.log(f"Ціну змінено з {old_price} на {self.price:.2f}")

# Демонстрація
print("✂ МІКСИНИ В ДІЇ")
print("=" = 70)

user = User("john_doe", "john@example.com")
user.update_email("john.doe@example.com")

print("\n📁 Дані користувача у вигляді словника:")
print(user.to_dict())

print("\n📦 Робота з товаром:")
product = Product("Ноутбук", 25000)
product.apply_discount(10)
print(product.to_json())

```

4. Поліморфізм та абстрактні класи

Поліморфізм – це здатність об’єктів різних класів відповідати на однакові виклики методів по-різному. Це одна з ключових концепцій ООП, яка дозволяє писати гнучкий код, що працює з різними типами об’єктів через спільний інтерфейс.

Поліморфізм без наслідування: качина типізація

В Python поліморфізм реалізується через **качину типізацію** (duck typing): "Якщо щось ходить як качка і крякає як качка, то це качка". Тобто важливо не те, до якого класу належить об’єкт, а те, які методи він має.

```

class Dog:
    def speak(self):
        return "Гав-гав!"

    def move(self):
        return "Біжить"

```

```

class Cat:
    def speak(self):
        return "Мяу!"

    def move(self):
        return "Крадеться"

class Bird:
    def speak(self):
        return "Чирик-чирик!"

    def move(self):
        return "Летить"

class Car:
    def move(self):
        return "Їде"
    # speak() відсутній!

def animal_sound(animal):
    """Функція, яка працює з будь-яким об'єктом, що має метод speak()"""
    return animal.speak()

def animal_movement(animal):
    """Функція, яка працює з будь-яким об'єктом, що має метод move()"""
    return animal.move()

# Демонстрація поліморфізму
animals = [Dog(), Cat(), Bird()]

print("🐾 ПОЛІМОРФІЗМ:")
for animal in animals:
    print(f" {animal.speak()} – {animal.move()}")

# Качина типізація в дії
print("\nВиклики функцій:")

```

```

for animal in animals:
    print(f" {animal_sound(animal)}")

# Це теж працює, хоча Car не є твариною!
car = Car()
print(f" {animal_movement(car)}")
# print(animal_sound(car)) # Помилка! Car не має методу speak()

```

Поліморфізм з наслідуванням

Наслідування дозволяє створювати ієрархії класів з однаковим інтерфейсом:

```

from math import pi

class Shape:
    """Абстрактний базовий клас для всіх фігур"""

    def area(self):
        """Метод має бути перевизначений у дочірніх класах"""
        raise NotImplementedError("Підкласи повинні реалізувати метод
area()")

    def perimeter(self):
        """Метод має бути перевизначений у дочірніх класах"""
        raise NotImplementedError("Підкласи повинні реалізувати метод
perimeter()")

    def __str__(self):
        return f"{self.__class__.__name__}: площа = {self.area():.2f}"

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return pi * self.radius ** 2

    def perimeter(self):

```

```
return 2 * pi * self.radius
```

```
class Rectangle(Shape):
```

```
    def __init__(self, width, height):
```

```
        self.width = width
```

```
        self.height = height
```

```
    def area(self):
```

```
        return self.width * self.height
```

```
    def perimeter(self):
```

```
        return 2 * (self.width + self.height)
```

```
class Triangle(Shape):
```

```
    def __init__(self, a, b, c):
```

```
        self.a = a
```

```
        self.b = b
```

```
        self.c = c
```

```
    def area(self):
```

```
        # Формула Герона
```

```
        s = (self.a + self.b + self.c) / 2
```

```
        return (s * (s - self.a) * (s - self.b) * (s - self.c)) ** 0.5
```

```
    def perimeter(self):
```

```
        return self.a + self.b + self.c
```

```
# Функція, яка демонструє поліморфізм
```

```
def print_shape_info(shape):
```

```
    print(f" {shape}")
```

```
    print(f" Периметр: {shape.perimeter():.2f}")
```

```
# Створюємо список різних фігур
```

```
shapes = [
```

```
    Circle(5),
```

```
    Rectangle(4, 6),
```

```
    Triangle(3, 4, 5)
```

]

```
print("📁 ПОЛІМОРФІЗМ З ФІГУРАМИ:")
for shape in shapes:
    print_shape_info(shape)
print()
```

Абстрактні класи

Абстрактний клас – це клас, який не може бути інстанційований (не можна створити його об’єкт). Він слугує шаблоном для інших класів і може містити абстрактні методи, які обов’язково мають бути реалізовані в дочірніх класах.

В Python абстрактні класи реалізуються за допомогою модуля abc (Abstract Base Classes).

```
from abc import ABC, abstractmethod
import math

class Vehicle(ABC):
    """Абстрактний базовий клас для транспортних засобів"""

    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    @abstractmethod
    def start(self):
        """Абстрактний метод – має бути реалізований у дочірніх класах"""
        pass

    @abstractmethod
    def stop(self):
        """Абстрактний метод – має бути реалізований у дочірніх класах"""
        pass

    @abstractmethod
    def fuel_consumption(self, distance):
        """Абстрактний метод – розрахунок витрати пального"""
        pass
```

```
def info(self):  
    """Звичайний метод – доступний у всіх дочірніх класах"""  
    return f"{self.brand} {self.model}"
```

Спроба створити об'єкт абстрактного класу
vehicle = Vehicle("Toyota", "Camry") # TypeError!

```
class Car(Vehicle):  
    def __init__(self, brand, model, fuel_efficiency):  
        super().__init__(brand, model)  
        self.fuel_efficiency = fuel_efficiency # л/100 км  
  
    def start(self):  
        return f"{self.info()}: Двигун запущено, вр-р-р-р!"  
  
    def stop(self):  
        return f"{self.info()}: Двигун зупинено"  
  
    def fuel_consumption(self, distance):  
        return (distance / 100) * self.fuel_efficiency
```

```
class ElectricCar(Vehicle):  
    def __init__(self, brand, model, battery_capacity, efficiency):  
        super().__init__(brand, model)  
        self.battery_capacity = battery_capacity # кВт·год  
        self.efficiency = efficiency # кВт·год/100 км  
  
    def start(self):  
        return f"{self.info()}: Електромотор запущено (тихо)"  
  
    def stop(self):  
        return f"{self.info()}: Електромотор зупинено"  
  
    def fuel_consumption(self, distance):  
        # Для електромобіля це витрата електроенергії  
        return (distance / 100) * self.efficiency
```



```

def charge(self, hours):
    return f"Заряджається {hours} годин"

class Bicycle(Vehicle):
    def __init__(self, brand, model):
        super().__init__(brand, model)

    def start(self):
        return f"{self.info()}: Починаю крутити педалі"

    def stop(self):
        return f"{self.info()}: Зупиняюся"

    def fuel_consumption(self, distance):
        return 0 # Велосипед не споживає пального!

# Демонстрація
print("🚗 АБСТРАКТНІ КЛАСИ")
print("==" = 70)

vehicles = [
    Car("Toyota", "Camry", 8.5),
    ElectricCar("Tesla", "Model 3", 75, 15),
    Bicycle("Giant", "Escape 3")
]

distance = 100
print(f"Подорож на {distance} км:\n")

for vehicle in vehicles:
    print(vehicle.start())
    consumption = vehicle.fuel_consumption(distance)
    if isinstance(vehicle, ElectricCar):
        print(f"Витрата енергії: {consumption:.1f} кВт·год")
    elif isinstance(vehicle, Bicycle):
        print(f"Витрата енергії: {consumption} – це ж велосипед!")

```

```
else:
    print(f" Витрата пального: {consumption:.1f} л")
print(vehicle.stop())
print()
```

Перевірка типів з абстрактними класами

```
from abc import ABC, abstractmethod
```

```
class Drawable(ABC):
    """Абстрактний клас для об'єктів, які можна малювати"""

    @abstractmethod
    def draw(self):
        pass
```

```
class Resizable(ABC):
    """Абстрактний клас для об'єктів, які можна змінювати розмір"""

    @abstractmethod
    def resize(self, factor):
        pass
```

```
class Circle(Drawable, Resizable):
    def __init__(self, radius):
        self.radius = radius

    def draw(self):
        return f"○ Коло радіусом {self.radius}"

    def resize(self, factor):
        self.radius *= factor
        return f"Розмір змінено, новий радіус: {self.radius}"
```

```
class Square(Drawable):
    def __init__(self, side):
        self.side = side
```

```

def draw(self):
    return f"□ Квадрат зі стороною {self.side}"

# resize не реалізовано, але це не помилка,
# бо Square не успадковує Resizable

def draw_all(objects):
    for obj in objects:
        if isinstance(obj, Drawable):
            print(obj.draw())
        else:
            print(f"{obj} не можна намалювати")

def resize_all(objects, factor):
    for obj in objects:
        if isinstance(obj, Resizable):
            print(obj.resize(factor))
        else:
            print(f"{obj} не підтримує зміну розміру")

# Демонстрація
print("🔍 ПЕРЕВІРКА ТИПІВ")
print("=" = 70)

circle = Circle(5)
square = Square(10)

objects = [circle, square, "рядок, який не можна намалювати"]

draw_all(objects)
print()
resize_all(objects, 2)

```

Практичний приклад: система плагінів

```

from abc import ABC, abstractmethod
import json
import csv

```

```

class DataExporter(ABC):
    """Абстрактний клас для експортерів даних"""

    @abstractmethod
    def export(self, data, filename):
        """Експортує дані у файл"""
        pass

    @abstractmethod
    def get_format_name(self):
        """Повертає назву формату"""
        pass

class JSONExporter(DataExporter):
    def export(self, data, filename):
        with open(filename, 'w', encoding='utf-8') as f:
            json.dump(data, f, ensure_ascii=False, indent=4)
        return f"Дані експортовано в {filename} (JSON)"

    def get_format_name(self):
        return "JSON"

class CSVExporter(DataExporter):
    def export(self, data, filename):
        if not data:
            return "Немає даних для експорту"

        with open(filename, 'w', newline="", encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=data[0].keys())
            writer.writeheader()
            writer.writerows(data)
        return f"Дані експортовано в {filename} (CSV)"

    def get_format_name(self):
        return "CSV"

```

```

class XMLExporter(DataExporter):
    def export(self, data, filename):
        # Спрощена імітація XML
        with open(filename, 'w', encoding='utf-8') as f:
            f.write('<?xml version="1.0" encoding="UTF-8"?>\n')
            f.write('<data>\n')
            for item in data:
                f.write(' <record>\n')
                for key, value in item.items():
                    f.write(f' <{key}>{value}</{key}>\n')
                f.write(' </record>\n')
            f.write('</data>')
        return f"Дані експортовано в {filename} (XML)"

    def get_format_name(self):
        return "XML"

class DataProcessor:
    """Клас для обробки даних з підтримкою різних форматів експорту"""

    def __init__(self):
        self.data = []
        self.exporters = {}

    def add_data(self, item):
        self.data.append(item)

    def register_exporter(self, exporter):
        """Реєструє новий експортер"""
        self.exporters[exporter.get_format_name().lower()] = exporter
        print(f"✓ Зареєстровано експортер для формату {exporter.get_format_name()}")

    def export_data(self, format_name, filename):
        """Екпортує дані у вказаному форматі"""
        exporter = self.exporters.get(format_name.lower())
        if not exporter:

```

```

    return f"✗ Формат {format_name} не підтримується"

    if not self.data:
        return "✗ Немає даних для експорту"

    return exporter.export(self.data, filename)

# Демонстрація
print("📦 СИСТЕМА ПЛАГІНІВ ЕКСПОРТУ")
print("==" = 70)

# Створюємо процесор даних
processor = DataProcessor()

# Додаємо дані
processor.add_data({"name": "Олена", "age": 19, "city": "Київ"})
processor.add_data({"name": "Іван", "age": 20, "city": "Львів"})
processor.add_data({"name": "Марія", "age": 18, "city": "Харків"})

# Реєструємо експортери
processor.register_exporter(JSONExporter())
processor.register_exporter(CSVExporter())
processor.register_exporter(XMLExporter())

# Експортуємо дані
print("\n⬆️ ЕКСПОРТ ДАНИХ:")
for format_name in ["json", "csv", "xml", "pdf"]:
    result = processor.export_data(format_name, f"data.{format_name}")
    print(f" {result}")

```

Підсумки

1. **Наслідування** дозволяє створювати ієрархії класів і повторно використовувати код
2. **super()** правильно викликає методи батьківських класів з урахуванням MRO
3. **Множинне наслідування** потужне, але вимагає розуміння MRO

4. **Міксини** – хороший спосіб використання множинного наслідування для додавання функціональності

5. **Поліморфізм** дозволяє працювати з різними об'єктами через спільний інтерфейс

6. **Качина типізація** – особливість Python, де важливі методи, а не типи

7. **Абстрактні класи** визначають інтерфейси і гарантують реалізацію методів

У наступній лекції ми розглянемо інкапсуляцію та спеціальні методи класів – як захищати дані та робити класи ще більш потужними.

Питання для обговорення

1. Поясніть концепцію наслідування в ООП. Навіщо воно потрібне? Наведіть приклад ієрархії класів з реального життя (наприклад, ТранспортнийЗасіб -> Автомобіль -> Електромобіль).

2. Проаналізуйте код:

```
class Animal:
    def speak(self):
        return "Тварина видає звук"
```

```
class Dog(Animal):
    def speak(self):
        return "Гав-гав!"
```

```
class Cat(Animal):
    def speak(self):
        return "Мяу!"
```

Що таке перевизначення методів (overriding)? Чому при виклику speak() для об'єктів Dog та Cat отримуємо різні результати?

3. Поясніть призначення функції super(). У яких випадках вона використовується? Наведіть приклад, де дочірній клас розширює метод батьківського класу, викликаючи його через super().

4. Напишіть клас Vehicle з атрибутами brand та model та методом info(). Створіть дочірній клас Car, який додає атрибут doors та розширює метод info(), щоб він виводив також кількість дверей.

5. Що таке множинне наслідування? Чому воно може бути небезпечним? Поясніть проблему ромба (diamond problem) і як Python її вирішує за допомогою MRO (Method Resolution Order).

6. Як подивитися порядок вирішення методів (MRO) для класу? Що таке атрибут `__mro__` і що він показує?

7. Поясніть концепцію поліморфізму. Як він проявляється у коді з попереднього прикладу (Animal, Dog, Cat)? Чому можна написати функцію, яка приймає будь-який об'єкт з методом `speak()`, і вона буде працювати?

8. Що таке "качина типізація" (duck typing) в Python? Поясніть фразу: "Якщо щось ходить як качка і крякає як качка, то це качка". Наведіть приклад.

9. Проаналізуйте код:

```
class Shape:
    def area(self):
        raise NotImplementedError("Підкласи мають реалізувати цей метод")
```

Чому в базовому класі Shape метод `area` кидає помилку? Як це пов'язано з абстрактними класами?

10. Що таке абстрактні класи в Python? Для чого використовується модуль `abc` та декоратор `@abstractmethod`? Напишіть абстрактний клас Shape з абстрактним методом `area` та створіть два конкретні класи Circle та Rectangle, які його реалізують.

ЛЕКЦІЯ 16

Інкапсуляція та спеціальні методи класів

План лекції

1. Принцип інкапсуляції: приховування внутрішньої реалізації.

Пояснимо, що таке інкапсуляція, навіщо вона потрібна. Розберемо конвенції з одним підкресленням `_` (protected) та двома підкресленнями `__` (private) для приховування атрибутів. Пояснимо механізм name mangling у Python.

2. Властивості (properties): контроль доступу до атрибутів.

Навчимося створювати геттери, сеттери та делетери за допомогою декоратора `@property`. Покажемо, як це дозволяє додавати логіку при доступі до атрибутів, зберігаючи простий синтаксис звернення через крапку.

3. Магічні методи для порівняння та арифметики.

Розширимо знання про магічні методи: методи порівняння (`__eq__`, `__lt__`, `__gt__`), арифметичні методи (`__add__`, `__sub__`, `__mul__`). Покажемо, як зробити класи "розумними" і працювати з ними як з вбудованими типами.

4. Магічні методи для контейнерів та контекстних менеджерів.

Розглянемо методи для створення власних контейнерів (`__len__`, `__getitem__`, `__setitem__`) та контекстних менеджерів (`__enter__`, `__exit__`), які дозволяють використовувати класи з оператором `with`.

1. Принцип інкапсуляції: приховування внутрішньої реалізації

Інкапсуляція – це один з трьох основних принципів ООП (разом з наслідуванням та поліморфізмом). Він означає об'єднання даних і методів роботи з цими даними в одному об'єкті, а також **приховування внутрішньої реалізації** від зовнішнього коду.

Навіщо потрібна інкапсуляція?

1. **Захист даних** – запобігає несанкціонованому доступу та зміні даних
2. **Модульність** – зміна внутрішньої реалізації не впливає на зовнішній код
3. **Контроль доступу** – можна додати логіку валідації при зміні даних
4. **Спрощення інтерфейсу** – користувач класу бачить тільки те, що йому потрібно

Рівні доступу в Python

На відміну від багатьох мов (Java, C++), Python не має жорстких модифікаторів доступу (private, protected, public). Замість цього використовуються **конвенції**:

- **Публічні атрибути** – без підкреслень: self.name
- **Захищені атрибути** – з одним підкресленням: self._age (конвенція: "не чіпай це ззовні")
- **Приватні атрибути** – з двома підкресленнями: self.__salary (механізм name mangling)

```
class Person:
    """Демонстрація рівнів доступу"""

    def __init__(self, name, age, salary):
        self.name = name          # публічний атрибут
        self._age = age           # захищений (protected) - конвенція
        self.__salary = salary    # приватний (private) - name mangling

    def get_salary(self):
        """Публічний метод для доступу до приватного атрибута"""
        return self.__salary

    def _internal_method(self):
        """Захищений метод (тільки для внутрішнього використання)"""
        return "Це внутрішній метод"

    def __private_method(self):
        """Приватний метод"""
        return "Це приватний метод"

# Створюємо об'єкт
p = Person("Олена", 19, 50000)

# Доступ до публічного атрибута
print(f"Публічний name: {p.name}")

# Доступ до захищеного атрибута (технічно можливий, але не
рекомендується)
```

```
print(f"Захищений _age: {p._age}") # Працює, але це поганий тон!
```

```
# Спроба доступу до приватного атрибута
```

```
# print(p.__salary) # AttributeError!
```

```
# Але Python "приховує" його, змінюючи ім'я
```

```
print(f"Прихований __salary: {p._Person__salary}") # Так можна, але не робіть так!
```

```
# Виклик методів
```

```
print(p._internal_method()) # Працює, але не рекомендується
```

```
# print(p.__private_method()) # AttributeError!
```

```
print(p._Person__private_method()) # Технічно можливо, але не робіть так!
```

Механізм name mangling

Коли ми використовуємо подвійне підкреслення __, Python автоматично змінює ім'я атрибута, додаючи перед ним `_Ім'яКласу`. Це ускладнює випадковий доступ ззовні, але не робить його неможливим.

```
class BankAccount:
```

```
    def __init__(self, owner, balance):
```

```
        self.owner = owner
```

```
        self.__balance = balance # приватний атрибут
```

```
    def deposit(self, amount):
```

```
        if amount > 0:
```

```
            self.__balance += amount
```

```
            return f"Поповнено. Новий баланс: {self.__balance}"
```

```
        return "Некоректна сума"
```

```
    def withdraw(self, amount):
```

```
        if 0 < amount <= self.__balance:
```

```
            self.__balance -= amount
```

```
            return f"Знято. Новий баланс: {self.__balance}"
```

```
        return "Недостатньо коштів або некоректна сума"
```

```
    def get_balance(self):
```

```
        return self.__balance
```

Створюємо рахунок

```
acc = BankAccount("Олена", 1000)
```

Подивимося, які атрибути є в об'єкта

```
print("Атрибути об'єкта:")
```

```
for attr in dir(acc):
```

```
    if not attr.startswith('__'):
```

```
        print(f" {attr}")
```

Доступ через публічний метод

```
print(f"\nБаланс через метод: {acc.get_balance()}")
```

"Прихований" доступ (тільки для демонстрації!)

```
print(f"Баланс через name mangling: {acc._BankAccount__balance}")
```

Чому варто дотримуватися конвенцій?

Навіть якщо технічно можливо отримати доступ до "приватних" атрибутів, дотримання конвенцій робить код кращим:

```
class Temperature:
```

```
    """Клас для роботи з температурою"""
```

```
    def __init__(self, celsius):
```

```
        self._celsius = celsius # захищений атрибут
```

```
    @property
```

```
    def celsius(self):
```

```
        """Геттер для температури"""
```

```
        return self._celsius
```

```
    @celsius.setter
```

```
    def celsius(self, value):
```

```
        """Сеттер з валідацією"""
```

```
        if value < -273.15:
```

```
            raise ValueError("Температура не може бути нижче абсолютного  
нуля")
```

```
        self._celsius = value
```

```

@property
def fahrenheit(self):
    """Обчислювана властивість"""
    return (self._celsius * 9/5) + 32

# Використання
temp = Temperature(25)
print(f"Цельсій: {temp.celsius}")
print(f"Фаренгейт: {temp.fahrenheit}")

temp.celsius = 30 # працює через сеттер
print(f"Нова температура: {temp.celsius}")

# temp.celsius = -300 # ValueError!

# Але ми все ще можемо отримати доступ до _celsius
print(f"Прямий доступ (не рекомендується): {temp._celsius}")

```

Практичний приклад: клас з інкапсуляцією

```

import re

class User:
    """Клас користувача з правильною інкапсуляцією"""

    def __init__(self, username, email, password):
        self.username = username
        self._email = None # ініціалізуємо через сеттер
        self.__password = None # приватний атрибут

        # Використовуємо сеттери для валідації
        self.email = email
        self._set_password(password)

        self._login_attempts = 0
        self._is_locked = False

```

```

@property
def email(self):
    """Геттер для email"""
    return self._email

@email.setter
def email(self, value):
    """Сеттер з валідацією"""
    if not re.match(r"^[^@]+@[^@]+\.[^@]+$", value):
        raise ValueError("Некоректний email")
    self._email = value

def _set_password(self, password):
    """Внутрішній метод для встановлення пароля"""
    if len(password) < 8:
        raise ValueError("Пароль має містити щонайменше 8 символів")
    if not any(c.isupper() for c in password):
        raise ValueError("Пароль має містити хоча б одну велику літеру")
    if not any(c.isdigit() for c in password):
        raise ValueError("Пароль має містити хоча б одну цифру")

    self.__password = self._hash_password(password)

def _hash_password(self, password):
    """Приватний метод для хешування пароля (спрощено)"""
    # У реальному житті тут був би складний хеш
    return "".join(str(ord(c)) for c in password)

def check_password(self, password):
    """Перевіряє пароль"""
    if self._is_locked:
        return False

    if self.__password == self._hash_password(password):
        self._login_attempts = 0
        return True

```

```

self._login_attempts += 1
if self._login_attempts >= 3:
    self._is_locked = True
    print("🔒 Користувача заблоковано через 3 невдалі спроби")

return False

def reset_password(self, old_password, new_password):
    """Зміна пароля"""
    if not self.check_password(old_password):
        return "❌ Невірний поточний пароль"

    try:
        self._set_password(new_password)
        return "✅ Пароль успішно змінено"
    except ValueError as e:
        return f"❌ {e}"

@property
def is_locked(self):
    return self._is_locked

def __str__(self):
    return f"User: {self.username}, Email: {self._email}, Locked: {self._is_locked}"

# Демонстрація
print("👤 СИСТЕМА КОРИСТУВАЧІВ")
print("==" = 70)

# Створюємо користувача
try:
    user = User("john_doe", "john@example.com", "Password123")
    print(user)

# Спроби входу

```

```

print("\n🔒 Спроби входу:")
print(f"Спроба 1 (wrong): {user.check_password('wrong')}")
print(f"Спроба 2 (wrong): {user.check_password('wrong')}")
print(f"Спроба 3 (wrong): {user.check_password('wrong')}")
print(f"Спроба 4 (правильний): {user.check_password('Password123')}")

# Зміна пароля
print("\n🔑 Зміна пароля:")
print(user.reset_password("Password123", "NewPass456"))
print(user.reset_password("Password123", "weak")) # занадто слабкий

except ValueError as e:
    print(f"Помилка: {e}")

```

2. Властивості (properties): контроль доступу до атрибутів

Декоратор `@property` – це елегантний спосіб створення геттерів, сеттерів та делетерів в Python. Він дозволяє додавати логіку при доступі до атрибутів, зберігаючи простий синтаксис звернення через крапку.

Чому не просто зробити методи `get_x()` та `set_x()`?

```

# Поганий стиль (Java-подібний)
class BadRectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    def get_width(self):
        return self._width

    def set_width(self, value):
        if value <= 0:
            raise ValueError("Ширина має бути додатною")
        self._width = value

```



```

def get_height(self):
    return self._height

def set_height(self, value):
    if value <= 0:
        raise ValueError("Висота має бути додатною")
    self._height = value

def get_area(self):
    return self._width * self._height

# Використання незручне
rect = BadRectangle(10, 5)
rect.set_width(15) # незручно!
print(rect.get_area())

```

Кращий підхід з property

```

class Rectangle:
    """Прямокутник з використанням property"""

    def __init__(self, width, height):
        self._width = width
        self._height = height

    @property
    def width(self):
        """Геттер для ширини"""
        return self._width

    @width.setter
    def width(self, value):
        """Сеттер для ширини з валідацією"""
        if value <= 0:
            raise ValueError("Ширина має бути додатною")
        self._width = value

    @property

```

```

def height(self):
    """Геттер для висоти"""
    return self._height

@height.setter
def height(self, value):
    """Сеттер для висоти з валідацією"""
    if value <= 0:
        raise ValueError("Висота має бути додатною")
    self._height = value

@property
def area(self):
    """Обчислювана властивість (тільки для читання)"""
    return self._width * self._height

@property
def perimeter(self):
    """Обчислювана властивість"""
    return 2 * (self._width + self._height)

```

Використання

```

rect = Rectangle(10, 5)
print(f"Ширина: {rect.width}")      # викликає геттер
print(f"Висота: {rect.height}")     # викликає геттер
print(f"Площа: {rect.area}")        # обчислювана властивість
print(f"Периметр: {rect.perimeter}")

```

```

rect.width = 15 # викликає сеттер
print(f"Нова площа: {rect.area}")

```

rect.width = -5 # ValueError!

rect.area = 100 # AttributeError (немає сеттера)

Повний приклад з геттером, сеттером і делетером

```

class Temperature:
    """Клас для роботи з температурою з різними одиницями виміру"""

```

```

def __init__(self, celsius=0):
    self._celsius = celsius
    self._unit = "C"

@property
def celsius(self):
    """Температура в градусах Цельсія"""
    return self._celsius

@celsius.setter
def celsius(self, value):
    if value < -273.15:
        raise ValueError("Температура не може бути нижче абсолютного
нуля")
    self._celsius = value
    self._unit = "C"

@celsius.deleter
def celsius(self):
    """Делетер (рідко використовується)"""
    print("Видалення температури")
    del self._celsius

@property
def fahrenheit(self):
    """Температура в градусах Фаренгейта"""
    return (self._celsius * 9/5) + 32

@fahrenheit.setter
def fahrenheit(self, value):
    """Встановлення температури через Фаренгейти"""
    self.celsius = (value - 32) * 5/9
    self._unit = "F"

@property
def kelvin(self):

```

```

        """Температура в Кельвінах"""
        return self._celsius + 273.15

    @kelvin.setter
    def kelvin(self, value):
        if value < 0:
            raise ValueError("Температура в Кельвінах не може бути
від'ємною")
        self.celsius = value - 273.15
        self._unit = "K"

    @property
    def unit(self):
        """Одиниця виміру, в якій була встановлена температура"""
        return self._unit

    def __str__(self):
        return f"{self._celsius:.1f}°C ({self.fahrenheit:.1f}°F, {self.kelvin:.1f}K)"

# Демонстрація
print(" 🐞 РОБОТА З ТЕМПЕРАТУРОЮ")
print("==" = 70)

temp = Temperature(25)
print(f"Початкова: {temp}")

temp.celsius = 30
print(f"Після зміни: {temp}")

temp.fahrenheit = 100
print(f"Після встановлення 100°F: {temp} (встановлено в {temp.unit})")

temp.kelvin = 300
print(f"Після встановлення 300K: {temp} (встановлено в {temp.unit})")

# temp.celsius = -300 # ValueError!

```

```
# Використання делетера
# del temp.celsius
# print(temp.celsius) # AttributeError
```

Обчислювані властивості з кешуванням

```
class Circle:
    """Коло з кешуванням обчислюваних властивостей"""

    def __init__(self, radius):
        self._radius = radius
        self._area = None
        self._circumference = None

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value <= 0:
            raise ValueError("Радіус має бути додатним")
        self._radius = value
        # Скидаємо кеш при зміні радіуса
        self._area = None
        self._circumference = None

    @property
    def area(self):
        """Площа кола (з кешуванням)"""
        if self._area is None:
            import math
            self._area = math.pi * self._radius ** 2
            print(" (обчислення площі...)")
        return self._area

    @property
    def circumference(self):
```

```

        """Довжина кола (з кешуванням)"""
        if self._circumference is None:
            import math
            self._circumference = 2 * math.pi * self._radius
            print(" (обчислення довжини...)")
        return self._circumference

# Демонстрація
print("\nО КОЛО З КЕШУВАННЯМ")
print("r = 70)

circle = Circle(5)
print(f"Радіус: {circle.radius}")

print("Перший запит площі:")
print(f"Площа: {circle.area:.2f}")

print("\nДругий запит площі (з кешу):")
print(f"Площа: {circle.area:.2f}")

print("\nЗміна радіуса...")
circle.radius = 10

print("Запит площі після зміни:")
print(f"Площа: {circle.area:.2f}")
print(f"Довжина кола: {circle.circumference:.2f}")

```

Практичний приклад: клас BankAccount з властивостями

```

from datetime import datetime
import logging

# Налаштовуємо логування
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(message)s')

class BankAccount:
    """Банківський рахунок з використанням властивостей"""

```

```

def __init__(self, account_number, owner, initial_balance=0):
    self._account_number = account_number
    self._owner = owner
    self._balance = initial_balance
    self._currency = "UAH"
    self._transactions = []
    self._frozen = False
    self._overdraft_limit = 0
    self._interest_rate = 0.01 # 1% річних

    self._log_transaction("Рахунок відкрито", initial_balance)

@property
def account_number(self):
    """Номер рахунку (тільки для читання)"""
    return self._account_number

@property
def owner(self):
    """Власник рахунку (тільки для читання)"""
    return self._owner

@property
def balance(self):
    """Поточний баланс"""
    return self._balance

@property
def currency(self):
    return self._currency

@property
def frozen(self):
    """Чи заморожений рахунок"""
    return self._frozen

```

```

@frozen.setter
def frozen(self, value):
    """Замороження/розмороження рахунку"""
    if not isinstance(value, bool):
        raise ValueError("frozen має бути булевим значенням")
    self._frozen = value
    status = "заморожено" if value else "розморожено"
    self._log_transaction(f"Рахунок {status}")

@property
def overdraft_limit(self):
    """Ліміт овердрафту"""
    return self._overdraft_limit

@overdraft_limit.setter
def overdraft_limit(self, value):
    """Встановлення ліміту овердрафту"""
    if value < 0:
        raise ValueError("Ліміт овердрафту не може бути від'ємним")
    self._overdraft_limit = value
    self._log_transaction(f"Встановлено ліміт овердрафту: {value}
{self.currency}")

@property
def available_balance(self):
    """Доступний баланс з урахуванням овердрафту"""
    return self._balance + self._overdraft_limit

@property
def transaction_count(self):
    """Кількість транзакцій"""
    return len(self._transactions)

@property
def last_transaction(self):
    """Остання транзакція"""
    if self._transactions:

```



```

        return self._transactions[-1]
    return None

def _log_transaction(self, description, amount=None):
    """Внутрішній метод для логування транзакцій"""
    transaction = {
        'timestamp': datetime.now(),
        'description': description,
        'balance': self._balance
    }
    if amount is not None:
        transaction['amount'] = amount
    self._transactions.append(transaction)
    logging.info(f"{self._account_number}: {description}")

def deposit(self, amount):
    """Поповнення рахунку"""
    if self._frozen:
        raise PermissionError("Рахунок заморожено")

    if amount <= 0:
        raise ValueError("Сума поповнення має бути додатною")

    self._balance += amount
    self._log_transaction(f"Поповнення", amount)
    return f"✓ Поповнено {amount} {self.currency}. Новий баланс: {self._balance}"

def withdraw(self, amount):
    """Зняття коштів"""
    if self._frozen:
        raise PermissionError("Рахунок заморожено")

    if amount <= 0:
        raise ValueError("Сума зняття має бути додатною")

    if amount > self.available_balance:

```

```

        raise ValueError(f"Недостатньо коштів. Доступно:
{self.available_balance} {self.currency}")

    self._balance -= amount
    self._log_transaction(f"Зняття", -amount)
    return f"✓ Знято {amount} {self.currency}. Новий баланс:
{self._balance}"

def transfer(self, other_account, amount):
    """Переказ на інший рахунок"""
    if not isinstance(other_account, BankAccount):
        raise TypeError("Отримувач має бути BankAccount")

    self.withdraw(amount)
    other_account.deposit(amount)
    self._log_transaction(f"Переказ на рахунок
{other_account.account_number}", -amount)
    return f"✓ Переказано {amount} {self.currency} на рахунок
{other_account.account_number}"

@property
def statement(self):
    """Виписка по рахунку"""
    lines = []
    lines.append("=" * 70)
    lines.append(f"ВИПИСКА ПО РАХУНКУ")
    lines.append("=" * 70)
    lines.append(f"Рахунок: {self._account_number}")
    lines.append(f"Власник: {self._owner}")
    lines.append(f"Баланс: {self._balance} {self._currency}")
    lines.append(f"Доступно: {self.available_balance} {self._currency}")
    lines.append(f"Статус: {'ЗАМОРОЖЕНО' if self._frozen else
'АКТИВНИЙ'}")
    lines.append("-" * 70)
    lines.append("ТРАНЗАКЦІЇ:")

    for t in self._transactions[-10:]: # останні 10 транзакцій

```

```

time_str = t['timestamp'].strftime("%Y-%m-%d %H:%M")
amount_str = f"{t.get('amount', '')} {self._currency}" if 'amount' in t
else ""

lines.append(f" {time_str} | {t['description']:30} | {amount_str:10} |
{t['balance']} {self._currency}")

lines.append("=" * 70)
return "\n".join(lines)

```

Демонстрація

```

print("🏠 БАНКІВСЬКИЙ РАХУНОК З ВЛАСТИВОСТЯМИ")
print("=" * 70)

```

Створюємо рахунки

```

acc1 = BankAccount("UA123456789", "Олена Петренко", 1000)
acc2 = BankAccount("UA987654321", "Іван Коваленко", 500)

```

Демонструємо властивості

```

print(f"Рахунок {acc1.account_number}, власник: {acc1.owner}")
print(f"Баланс: {acc1.balance} {acc1.currency}")
print(f"Доступно: {acc1.available_balance} {acc1.currency}")

```

Операції

```

print("\n" + acc1.deposit(500))
print(acc1.withdraw(200))
print(acc1.transfer(acc2, 300))

```

Зміна ліміту овердрафту

```

acc1.overdraft_limit = 1000
print(f"\nНовий ліміт овердрафту: {acc1.overdraft_limit}")
print(f"Доступний баланс: {acc1.available_balance}")

```

Спроба зняти більше

```

try:
    acc1.withdraw(2000)
except ValueError as e:
    print(f"❌ {e}")

```

```

# Виписка
print("\n" + acc1.statement)

# Замороження рахунку
acc1.frozen = True
print(f"\nРахунок заморожено: {acc1.frozen}")

try:
    acc1.deposit(100)
except PermissionError as e:
    print(f"✗ {e}")

```

Підсумки

- **Інкапсуляція** – приховування внутрішньої реалізації класу
- Python використовує **конвенції** замість жорстких модифікаторів доступу
- Одне підкреслення `_` – "захищений" (protected) – конвенція
- Два підкреслення `__` – "приватний" (private) – name mangling
- `@property` дозволяє створювати геттери, сеттери та делетери
- Властивості зберігають простий синтаксис доступу через крапку
- Обчислювані властивості можуть мати кешування для оптимізації

У наступному пункті ми розглянемо магічні методи для порівняння та арифметики.

3. Магічні методи для порівняння та арифметики

Магічні методи (dunder methods – від double underscore) дозволяють визначити, як об'єкти класу поведуться при виконанні різних операцій. Вони викликаються автоматично при використанні відповідних операторів.

Методи порівняння

Python дозволяє визначити поведінку об'єктів при порівнянні за допомогою наступних методів:

Метод	Оператор	Опис
<code>__eq__(self, other)</code>	<code>==</code>	Рівність

Метод	Оператор	Опис
<code>__ne__(self, other)</code>	<code>!=</code>	Нерівність
<code>__lt__(self, other)</code>	<code><</code>	Менше
<code>__le__(self, other)</code>	<code><=</code>	Менше або дорівнює
<code>__gt__(self, other)</code>	<code>></code>	Більше
<code>__ge__(self, other)</code>	<code>>=</code>	Більше або дорівнює

```
class Person:
```

```
    """Клас людини з можливістю порівняння за віком"""
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
    def __eq__(self, other):
```

```
        """Перевірка на рівність (==)"""
```

```
        if not isinstance(other, Person):
```

```
            return NotImplemented
```

```
        return self.age == other.age
```

```
    def __ne__(self, other):
```

```
        """Перевірка на нерівність (!=)"""
```

```
        if not isinstance(other, Person):
```

```
            return NotImplemented
```

```
        return self.age != other.age
```

```
    def __lt__(self, other):
```

```
        """Менше (<)"""
```

```
        if not isinstance(other, Person):
```

```
            return NotImplemented
```

```
        return self.age < other.age
```

```
    def __le__(self, other):
```

```

        """Менше або дорівнює (<=)"""
        if not isinstance(other, Person):
            return NotImplemented
        return self.age <= other.age

    def __gt__(self, other):
        """Більше (>)"""
        if not isinstance(other, Person):
            return NotImplemented
        return self.age > other.age

    def __ge__(self, other):
        """Більше або дорівнює (>=)"""
        if not isinstance(other, Person):
            return NotImplemented
        return self.age >= other.age

    def __str__(self):
        return f"{self.name} ({self.age} років)"

# Демонстрація
print("👤 ПОРІВНЯННЯ ЛЮДЕЙ")
print("== = 70)

p1 = Person("Олена", 19)
p2 = Person("Іван", 20)
p3 = Person("Марія", 19)

print(f"p1: {p1}")
print(f"p2: {p2}")
print(f"p3: {p3}\n")

print(f"p1 == p2: {p1 == p2}")
print(f"p1 == p3: {p1 == p3}")
print(f"p1 != p2: {p1 != p2}")
print(f"p1 < p2: {p1 < p2}")
print(f"p1 > p2: {p1 > p2}")

```

```

print(f"p1 <= p3: {p1 <= p3}")
print(f"p2 >= p3: {p2 >= p3}")

# Сортування списку людей
people = [p1, p2, p3]
people.sort()
print("\nВідсортовано за віком:")
for p in people:
    print(f" {p}")

```

Спрощення за допомогою `functools.total_ordering`

Якщо визначити `__eq__` та один з методів порівняння (наприклад, `__lt__`), декоратор `@total_ordering` автоматично згенерує решту:

```

from functools import total_ordering

@total_ordering
class Book:
    """Клас книги з автоматичним створенням всіх методів порівняння"""

    def __init__(self, title, author, year, price):
        self.title = title
        self.author = author
        self.year = year
        self.price = price

    def __eq__(self, other):
        """Рівність (потрібно визначити обов'язково)"""
        if not isinstance(other, Book):
            return NotImplemented
        return (self.title, self.author, self.year) == (other.title, other.author,
other.year)

    def __lt__(self, other):
        """Менше (потрібно визначити обов'язково)"""
        if not isinstance(other, Book):
            return NotImplemented
        # Спочатку за роком, потім за ціною

```

```

    return (self.year, self.price) < (other.year, other.price)

def __str__(self):
    return f'“{self.title}” ({self.year}) - {self.price} грн"

# Демонстрація
print("\n📖 ПОРІВНЯННЯ КНИГ")
print("=" = 70)

books = [
    Book("Майстер і Маргарита", "М. Булгаков", 1967, 350),
    Book("Злочин і кара", "Ф. Достоевський", 1866, 420),
    Book("Війна і мир", "Л. Толстой", 1869, 500),
    Book("Енеїда", "І. Котляревський", 1798, 280)
]

print("Всі книги:")
for book in books:
    print(f" {book}")

print("\nВідсортовані книги (за роком, потім ціною):")
for book in sorted(books):
    print(f" {book}")

```

Арифметичні методи

Python дозволяє перевантажувати арифметичні оператори:

Метод	Оператор	Опис
<code>__add__(self, other)</code>	<code>+</code>	Додавання
<code>__sub__(self, other)</code>	<code>-</code>	Віднімання
<code>__mul__(self, other)</code>	<code>*</code>	Множення
<code>__truediv__(self, other)</code>	<code>/</code>	Ділення
<code>__floordiv__(self, other)</code>	<code>//</code>	Цілочисельне ділення

Метод	Оператор	Опис
<code>__mod__(self, other)</code>	<code>%</code>	Остача від ділення
<code>__pow__(self, other)</code>	<code>**</code>	Піднесення до степеня
<code>__radd__(self, other)</code>		Додавання зліва (коли лівий операнд не підтримує операцію)
<code>__iadd__(self, other)</code>	<code>+=</code>	Додавання з присвоєнням

```
class Vector:
```

```
    """Вектор у 2D просторі з арифметичними операціями"""
```

```
    def __init__(self, x, y):
```

```
        self.x = x
```

```
        self.y = y
```

```
    def __add__(self, other):
```

```
        """Додавання векторів: v1 + v2"""
```

```
        if isinstance(other, Vector):
```

```
            return Vector(self.x + other.x, self.y + other.y)
```

```
        elif isinstance(other, (int, float)):
```

```
            # Додавання скаляра до вектора
```

```
            return Vector(self.x + other, self.y + other)
```

```
        return NotImplemented
```

```
    def __radd__(self, other):
```

```
        """Додавання зліва: скаляр + вектор"""
```

```
        return self.__add__(other)
```

```
    def __sub__(self, other):
```

```
        """Віднімання векторів"""
```

```
        if isinstance(other, Vector):
```

```
            return Vector(self.x - other.x, self.y - other.y)
```

```
        return NotImplemented
```

```

def __mul__(self, other):
    """Множення вектора на скаляр"""
    if isinstance(other, (int, float)):
        return Vector(self.x * other, self.y * other)
    return NotImplemented

def __rmul__(self, other):
    """Множення зліва: скаляр * вектор"""
    return self.__mul__(other)

def __matmul__(self, other):
    """Скалярний добуток (оператор @)"""
    if isinstance(other, Vector):
        return self.x * other.x + self.y * other.y
    return NotImplemented

def __neg__(self):
    """Унарний мінус: -v"""
    return Vector(-self.x, -self.y)

def __abs__(self):
    """Модуль вектора (довжина)"""
    return (self.x ** 2 + self.y ** 2) ** 0.5

def __str__(self):
    return f"Vector({self.x}, {self.y})"

# Демонстрація
print("\n📐 АРИФМЕТИКА ВЕКТОРІВ")
print("==" = 70)

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

print(f"v1 = {v1}")
print(f"v2 = {v2}")

```

```

print(f"v1 + v2 = {v1 + v2}")
print(f"v1 - v2 = {v1 - v2}")
print(f"v1 * 2 = {v1 * 2}")
print(f"3 * v1 = {3 * v1}")
print(f"v1 @ v2 = {v1 @ v2}") # скалярний добуток
print(f"-v1 = {-v1}")
print(f"|v1| = {abs(v1):.2f}")

```

Практичний приклад: клас Money для роботи з грошима

```

from functools import total_ordering

@total_ordering
class Money:
    """Клас для роботи з грошовими сумами в різних валютах"""

    def __init__(self, amount, currency="UAH"):
        self.amount = float(amount)
        self.currency = currency

    # Конвертаційні курси (спрощено)
    _exchange_rates = {
        "UAH": 1,
        "USD": 38.5,
        "EUR": 41.2,
        "GBP": 48.3
    }

    def _convert_to(self, target_currency):
        """Конвертує суму в іншу валюту"""
        if self.currency == target_currency:
            return self.amount

        # Спочатку конвертуємо в UAH, потім в target
        in_uah = self.amount * self._exchange_rates[self.currency]
        return in_uah / self._exchange_rates[target_currency]

    def __add__(self, other):

```

```

        """Додавання грошових сум"""
    if isinstance(other, Money):
        # Якщо валюти різні, конвертуємо у валюту першого операнда
        other_amount = other._convert_to(self.currency)
        return Money(self.amount + other_amount, self.currency)
    elif isinstance(other, (int, float)):
        return Money(self.amount + other, self.currency)
    return NotImplemented

def __sub__(self, other):
    """Віднімання грошових сум"""
    if isinstance(other, Money):
        other_amount = other._convert_to(self.currency)
        return Money(self.amount - other_amount, self.currency)
    elif isinstance(other, (int, float)):
        return Money(self.amount - other, self.currency)
    return NotImplemented

def __mul__(self, other):
    """Множення на число"""
    if isinstance(other, (int, float)):
        return Money(self.amount * other, self.currency)
    return NotImplemented

def __rmul__(self, other):
    """Множення на число зліва"""
    return self.__mul__(other)

def __truediv__(self, other):
    """Ділення на число"""
    if isinstance(other, (int, float)):
        if other == 0:
            raise ZeroDivisionError("Ділення на нуль")
        return Money(self.amount / other, self.currency)
    return NotImplemented

def __eq__(self, other):

```

```

        """Порівняння на рівність"""
        if not isinstance(other, Money):
            return NotImplemented
        # Порівнюємо в UAH для точності
        return abs(self._convert_to("UAH") - other._convert_to("UAH")) < 0.01

    def __lt__(self, other):
        """Порівняння менше"""
        if not isinstance(other, Money):
            return NotImplemented
        return self._convert_to("UAH") < other._convert_to("UAH")

    def __str__(self):
        return f"{self.amount:.2f} {self.currency}"

    def __repr__(self):
        return f"Money({self.amount:.2f}, '{self.currency}')"

    def convert_to(self, target_currency):
        """Повертає новий об'єкт Money в іншій валюті"""
        new_amount = self._convert_to(target_currency)
        return Money(new_amount, target_currency)

# Демонстрація
print("💰 РОБОТА З ГРОШИМА")
print("== 70)

salary = Money(30000, "UAH")
bonus = Money(500, "USD")
print(f"Зарплата: {salary}")
print(f"Премія: {bonus}")

total = salary + bonus
print(f"Разом: {total}")

print(f"Разом в USD: {total.convert_to('USD')}")
print(f"Разом в EUR: {total.convert_to('EUR')}")

```

Арифметика

```
print(f"\nЗарплата * 2: {salary * 2}")
print(f"Зарплата / 2: {salary / 2}")
```

Порівняння

```
price1 = Money(1000, "USD")
price2 = Money(38000, "UAH")
print(f"\n{price1} == {price2}? {price1 == price2}")
print(f"{price1} < {price2}? {price1 < price2}")
```

Сортування

```
prices = [
    Money(100, "USD"),
    Money(3000, "UAH"),
    Money(50, "EUR"),
    Money(150, "GBP")
]

print("\nВідсортовані ціни (в еквіваленті):")
for price in sorted(prices):
    print(f" {price}")
```

4. Магічні методи для контейнерів та контекстних менеджерів

Методи для роботи з контейнерами

Ці методи дозволяють створювати класи, які поведуться як списки, словники або інші колекції:

Метод	Опис
<code>__len__(self)</code>	Повертає довжину (викликається <code>len(obj)</code>)
<code>__getitem__(self, key)</code>	Доступ за індексом/ключем (<code>obj[key]</code>)
<code>__setitem__(self, key, value)</code>	Встановлення значення (<code>obj[key] = value</code>)
<code>__delitem__(self, key)</code>	Видалення елемента (<code>del obj[key]</code>)
<code>__contains__(self, item)</code>	Перевірка належності (<code>item in obj</code>)

<code>__iter__(self)</code>	Повертає ітератор (for x in obj)

```

class CustomList:
    """Власна реалізація списку з магічними методами"""

    def __init__(self, items=None):
        self._items = list(items) if items else []

    def __len__(self):
        """Повертає довжину списку"""
        return len(self._items)

    def __getitem__(self, index):
        """Доступ за індексом"""
        if isinstance(index, slice):
            # Підтримка зрізів
            return CustomList(self._items[index])
        return self._items[index]

    def __setitem__(self, index, value):
        """Встановлення значення за індексом"""
        self._items[index] = value

    def __delitem__(self, index):
        """Видалення елемента за індексом"""
        del self._items[index]

    def __contains__(self, item):
        """Перевірка наявності елемента"""
        return item in self._items

    def __iter__(self):
        """Повертає ітератор"""
        return iter(self._items)

    def __reversed__(self):
        """Зворотний ітератор"""

```

```

    return reversed(self._items)

def append(self, item):
    """Додавання елемента"""
    self._items.append(item)

def __str__(self):
    return str(self._items)

# Демонстрація
print(" ВЛАСНИЙ СПИСОК")
print("=" = 70)

cl = CustomList([1, 2, 3, 4, 5])
print(f"Список: {cl}")
print(f"Довжина: {len(cl)}")
print(f"Елемент за індексом 2: {cl[2]}")
print(f"Зріз [1:4]: {cl[1:4]}")

cl[2] = 99
print(f"Після зміни: {cl}")

print(f"Чи є 99? {99 in cl}")
print(f"Чи є 100? {100 in cl}")

print("Ітерація:")
for item in cl:
    print(f" {item}", end=" ")
print()

print(f"Зворотна ітерація:")
for item in reversed(cl):
    print(f" {item}", end=" ")
print()

```

Приклад: власний словник з валідацією

```

class ValidatedDict:
    """Словник, який валідує ключі та значення"""

```



```

def __init__(self):
    self._data = {}

def __setitem__(self, key, value):
    """Встановлення значення з валідацією"""
    if not isinstance(key, str):
        raise TypeError("Ключ має бути рядком")

    if len(key) < 3:
        raise ValueError("Ключ має містити щонайменше 3 символи")

    if value is None:
        raise ValueError("Значення не може бути None")

    self._data[key] = value
    print(f"✔ Встановлено {key} = {value}")

def __getitem__(self, key):
    """Отримання значення"""
    if key not in self._data:
        raise KeyError(f"Ключ '{key}' не знайдено")
    return self._data[key]

def __delitem__(self, key):
    """Видалення ключа"""
    if key in self._data:
        del self._data[key]
        print(f"🗑 Видалено ключ '{key}'")

def __contains__(self, key):
    """Перевірка наявності ключа"""
    return key in self._data

def __len__(self):
    return len(self._data)

```

```

def __iter__(self):
    return iter(self._data)

def keys(self):
    return self._data.keys()

def values(self):
    return self._data.values()

def items(self):
    return self._data.items()

def __str__(self):
    return str(self._data)

# Демонстрація
print("\n ВАЛІДОВАНИЙ СЛОВНИК")
print("==" = 70)

vd = ValidatedDict()

try:
    vd["ab"] = 10 # занадто короткий ключ
except ValueError as e:
    print(f"Помилка: {e}")

vd["name"] = "Олена"
vd["age"] = 19
vd["city"] = "Київ"

print(f"\nВміст словника: {vd}")
print(f"Ім'я: {vd['name']}")
print(f"Вік: {vd['age']}")
print(f"Чи є ключ 'name'? {'name' in vd}")

del vd["city"]
print(f"Після видалення: {vd}")

```

```
print("\nІтерація по ключах:")
for key in vd:
    print(f" {key}: {vd[key]}")
```

Контекстні менеджери (__enter__ та __exit__)

Контекстні менеджери дозволяють використовувати об'єкти з оператором with. Вони гарантують правильне звільнення ресурсів навіть при виникненні помилок.

```
class FileManager:
```

```
    """Власний контекстний менеджер для роботи з файлами"""
```

```
    def __init__(self, filename, mode='r', encoding='utf-8'):
```

```
        self.filename = filename
```

```
        self.mode = mode
```

```
        self.encoding = encoding
```

```
        self.file = None
```

```
    def __enter__(self):
```

```
        """Викликається при вході в блок with"""
```

```
        print(f" Відкриваємо файл {self.filename}")
```

```
        self.file = open(self.filename, self.mode, encoding=self.encoding)
```

```
        return self.file
```

```
    def __exit__(self, exc_type, exc_val, exc_tb):
```

```
        """Викликається при виході з блоку with"""
```

```
        if self.file:
```

```
            self.file.close()
```

```
            print(f" Закриваємо файл {self.filename}")
```

```
        # Обробка помилок
```

```
        if exc_type:
```

```
            print(f" Сталася помилка: {exc_val}")
```

```
            # Повертаємо True, якщо помилку оброблено
```

```
            # False - якщо помилку треба передати далі
```

```
            return True # Помилка оброблена, програма продовжить роботу
```

```
        return True
```

Демонстрація

```
print(" КОНТЕКСТНИЙ МЕНЕДЖЕР")
print("=" = 70)
```

Створюємо тестовий файл

```
with open("test.txt", "w", encoding="utf-8") as f:
    f.write("Перший рядок\n")
    f.write("Другий рядок\n")
    f.write("Третій рядок\n")
```

Використовуємо наш контекстний менеджер

```
print("\nЧитання файлу:")
with FileManager("test.txt", "r") as f:
    content = f.read()
    print(f'Вміст файлу:\n{content}')

print("\nСпроба відкрити неіснуючий файл:")
with FileManager("nonexistent.txt", "r") as f:
    content = f.read() # Тут буде помилка
    print(content)

print("\nПрограма продовжує роботу!")
```

Приклад: контекстний менеджер для вимірювання часу

```
import time
```

```
class Timer:
```

```
    """Контекстний менеджер для вимірювання часу виконання"""
```

```
    def __init__(self, name="Блок коду"):
        self.name = name
```

```
    def __enter__(self):
        self.start = time.time()
        print(f"□ Початок виконання: {self.name}")
        return self
```

```

def __exit__(self, exc_type, exc_val, exc_tb):
    self.end = time.time()
    self.duration = self.end - self.start
    print(f"□ Кінець виконання: {self.name}")
    print(f" Час виконання: {self.duration:.4f} секунд")

    if exc_type:
        print(f" Сталася помилка: {exc_val}")
        return False # передаємо помилку далі

# Демонстрація
print(" ВИМІРЮВАННЯ ЧАСУ")
print("=" = 70)

with Timer("Сума чисел") as t:
    total = sum(range(1000000))
    print(f"Сума: {total}")

print()

with Timer("Пауза 1 секунда"):
    time.sleep(1)

print()

try:
    with Timer("Блок з помилкою"):
        x = 10 / 0 # Помилка!
except ZeroDivisionError:
    print("Помилка спіймана зовні")

Практичний приклад: клас DatabaseConnection
class DatabaseConnection:
    """Симуляція підключення до бази даних з контекстним менеджером"""

    def __init__(self, host, database, user, password):
        self.host = host
        self.database = database

```

```

self.user = user
self.password = password
self.connection = None
self.transaction_active = False

def __enter__(self):
    """Встановлення з'єднання при вході в with"""
    print(f" Підключення до {self.database} на {self.host}...")
    # Симулюємо підключення
    self.connection = f"Connection-{id(self)}"
    self.transaction_active = False
    print(f" Підключено!")
    return self

def __exit__(self, exc_type, exc_val, exc_tb):
    """Закриття з'єднання при виході з with"""
    if self.transaction_active and exc_type is None:
        print(" Транзакція не закрита, виконуємо rollback")
        self.rollback()

    print(f" Закриття з'єднання...")
    self.connection = None
    print(f" З'єднання закрито")

    if exc_type:
        print(f" Помилка під час роботи: {exc_val}")
        return False # передаємо помилку далі

    return True

def begin_transaction(self):
    """Початок транзакції"""
    if not self.connection:
        raise RuntimeError("Немає з'єднання")
    self.transaction_active = True
    print(" Почато транзакцію")

```

```
def commit(self):
    """Підтвердження транзакції"""
    if not self.transaction_active:
        raise RuntimeError("Немає активної транзакції")
    print("Транзакцію підтверджено (COMMIT)")
    self.transaction_active = False
```

```
def rollback(self):
    """Відміна транзакції"""
    if not self.transaction_active:
        raise RuntimeError("Немає активної транзакції")
    print("↩ Транзакцію скасовано (ROLLBACK)")
    self.transaction_active = False
```

```
def execute(self, query):
    """Виконання SQL-запиту"""
    if not self.connection:
        raise RuntimeError("Немає з'єднання")
    print(f"Виконання: {query}")
    return f"Результат запиту '{query}'"
```

Демонстрація

```
print("РОБОТА З БАЗОЮ ДАНИХ")
print("=" = 70)
```

Успішний сценарій

```
print("\n УСПІШНИЙ СЦЕНАРІЙ:")
with DatabaseConnection("localhost", "test_db", "user", "pass") as db:
    db.begin_transaction()
    db.execute("INSERT INTO users (name) VALUES ('Олена')")
    db.execute("INSERT INTO users (name) VALUES ('Іван')")
    db.commit()
```

Сценарій з помилкою

```
print("\n СЦЕНАРІЙ З ПОМИЛКОЮ:")
try:
    with DatabaseConnection("localhost", "test_db", "user", "pass") as db:
```

```

db.begin_transaction()
db.execute("INSERT INTO users (name) VALUES ('Марія')")
# Тут станеться помилка
x = 10 / 0
db.commit()
except ZeroDivisionError:
    print("Помилку спіймано в коді")

# Сценарій без явної транзакції
print("\n СЦЕНАРІЙ БЕЗ ТРАНЗАКЦІЇ:")
with DatabaseConnection("localhost", "test_db", "user", "pass") as db:
    db.execute("SELECT * FROM users")

```

Підсумковий приклад: клас Matrix з усіма магічними методами

```

class Matrix:
    """Клас для роботи з матрицями з повною підтримкою магічних
    методів"""

    def __init__(self, data):
        self._data = [list(row) for row in data]
        self.rows = len(self._data)
        self.cols = len(self._data[0]) if self.rows > 0 else 0

    # === Методи для контейнерів ===

    def __len__(self):
        """Кількість рядків"""
        return self.rows

    def __getitem__(self, index):
        """Доступ до рядка або елемента"""
        if isinstance(index, tuple):
            # matrix[i, j]
            i, j = index
            return self._data[i][j]
        # matrix[i]
        return self._data[index]

    def __setitem__(self, index, value):

```



```

        """Встановлення значення"""
        if isinstance(index, tuple):
            i, j = index
            self._data[i][j] = value
        else:
            self._data[index] = value

    def __iter__(self):
        """Ітерація по рядках"""
        return iter(self._data)

# === Арифметичні методи ===
    def __add__(self, other):
        """Додавання матриць"""
        if isinstance(other, Matrix):
            if self.rows != other.rows or self.cols != other.cols:
                raise ValueError("Матриці мають різні розміри")

            result = [
                [self._data[i][j] + other._data[i][j] for j in range(self.cols)]
                for i in range(self.rows)
            ]
            return Matrix(result)
        return NotImplemented

    def __sub__(self, other):
        """Віднімання матриць"""
        if isinstance(other, Matrix):
            if self.rows != other.rows or self.cols != other.cols:
                raise ValueError("Матриці мають різні розміри")

            result = [
                [self._data[i][j] - other._data[i][j] for j in range(self.cols)]
                for i in range(self.rows)
            ]
            return Matrix(result)
        return NotImplemented

```

```

def __mul__(self, other):
    """Множення на число або матрицю"""
    if isinstance(other, (int, float)):
        # Множення на скаляр
        result = [
            [self._data[i][j] * other for j in range(self.cols)]
            for i in range(self.rows)
        ]
        return Matrix(result)
    elif isinstance(other, Matrix):
        # Множення матриць
        if self.cols != other.rows:
            raise ValueError(f"Неможливо перемножити матриці {self.rows}x{self.cols} та {other.rows}x{other.cols}")

        result = [
            [
                sum(self._data[i][k] * other._data[k][j] for k in range(self.cols))
                for j in range(other.cols)
            ]
            for i in range(self.rows)
        ]
        return Matrix(result)
    return NotImplemented

def __rmul__(self, other):
    """Множення на число зліва"""
    return self.__mul__(other)

# === Методи порівняння ===
def __eq__(self, other):
    if not isinstance(other, Matrix):
        return False
    return self._data == other._data

def __str__(self):

```

```

max_width = max(len(str(val)) for row in self._data for val in row)
rows = []
for row in self._data:
    rows.append("[" + " ".join(f"{val:>{max_width}}" for val in row) + "]")
return "\n".join(rows)

def __repr__(self):
    return f"Matrix({self._data})"

# Демонстрація
print(" РОБОТА З МАТРИЦЯМИ")
print("=" = 70)

m1 = Matrix([[1, 2], [3, 4]])
m2 = Matrix([[5, 6], [7, 8]])
print("Матриця m1:")
print(m1)
print("\nМатриця m2:")
print(m2)

print("\nm1 + m2:")
print(m1 + m2)

print("\nm1 * 2:")
print(m1 * 2)

print("\nm1 * m2:")
print(m1 * m2)

print(f"\nРозмір m1: {len(m1)}x{len(m1[0])}")
print(f"Елемент m1[0,1]: {m1[0,1]}")

m1[0,1] = 99
print("\nПісля зміни m1[0,1]:")
print(m1)

```

Підсумки

1. **Магічні методи порівняння** дозволяють використовувати оператори `==`, `<`, `>` тощо

2. `@total_ordering` автоматично генерує всі методи порівняння, якщо визначені `__eq__` та один з інших

3. **Арифметичні магічні методи** (`__add__`, `__sub__`, `__mul__`) роблять об'єкти "математичними"

4. **Методи для контейнерів** (`__getitem__`, `__setitem__`, `__len__`) дозволяють працювати з об'єктами як зі списками

5. **Контекстні менеджери** (`__enter__`, `__exit__`) забезпечують безпечну роботу з ресурсами

Це завершує нашу подорож основами ООП в Python. У наступних лекціях ми розглянемо графічні бібліотеки Turtle та Matplotlib, а також створення GUI з Tkinter.

Питання для обговорення

1. Поясніть принцип інкапсуляції в ООП. Чому важливо приховувати внутрішню реалізацію класу від зовнішнього коду? Які переваги це дає?

2. Як в Python реалізується інкапсуляція? Поясніть різницю між публічними атрибутами, захищеними (з одним підкресленням `_`) та приватними (з двома підкресленнями `__`). Чи можна отримати доступ до приватних атрибутів ззовні?

3. Проаналізуйте код:

```
class Person:
    def __init__(self, name, age, salary):
        self.name = name
        self._age = age
        self.__salary = salary
```

```
p = Person("Олена", 19, 50000)
print(p.name)
print(p._age)
print(p.__salary)
```

Які рядки викличуть помилку? Чому? Як все ж таки отримати доступ до `__salary` (навіть якщо цього робити не варто)?

4. Що таке name mangling в Python? Як Python змінює імена приватних атрибутів? Чому це робиться, якщо доступ все одно можна отримати?

5. Поясніть призначення декоратора @property. Які переваги він дає порівняно зі створенням окремих методів get_age() та set_age()?

6. Напишіть клас Temperature з приватним атрибутом _celsius. Додайте властивість celsius з геттером та сеттером. Сеттер повинен перевіряти, що температура не нижче абсолютного нуля (-273.15°C). Додайте також властивість fahrenheit, яка обчислюється на основі celsius.

7. Проаналізуйте код:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius
        self._area = None

    @property
    def area(self):
        if self._area is None:
            self._area = 3.14 * self._radius ** 2
        return self._area
```

Що таке кешування обчислюваних властивостей? Чому воно корисне? Коли значення _area має бути скинуте?

8. Поясніть призначення методу __str__. Чим він відрізняється від __repr__? В яких ситуаціях викликається кожен з них? Наведіть приклад класу, де ці методи повертають різні рядки.

9. Напишіть клас Vector2D з магічними методами __add__, __sub__, __mul__ (для множення на число) та __abs__ (для обчислення довжини вектора). Продемонструйте їх роботу.

10. Що таке контекстні менеджери і для чого вони потрібні? Які методи потрібно реалізувати, щоб клас можна було використовувати з оператором with? Напишіть простий контекстний менеджер для вимірювання часу виконання блоку коду.

ЛЕКЦІЯ 17

Графічна бібліотека Turtle та візуалізація даних з Matplotlib

План лекції

1. Вступ до Turtle: перші кроки в графіці.

Познайомимося з модулем turtle, навчимося створювати черепашку та керувати нею. Розберемо базові команди руху: вперед, назад, повороти. Покажемо, як налаштовувати колір та товщину лінії.

2. Малювання геометричних фігур та патернів за допомогою циклів.

Навчимося малювати квадрати, кола, трикутники, використовуючи цикли для автоматизації. Покажемо, як створювати складні патерни, квіти, спіралі. Розберемо роботу з кольором та заповненням фігур.

3. Система координат, фрактали та рекурсія в Turtle.

Розберемо систему координат у вікні Turtle, навчимося позиціонувати черепашку абсолютно. Покажемо, як малювати фрактали (дерево, сніжинку Коха) за допомогою рекурсії. Це чудова демонстрація потужності рекурсії.

4. Вступ до Matplotlib: перша візуалізація даних.

Перейдемо до професійної бібліотеки для візуалізації. Навчимося створювати лінійні графіки, налаштовувати осі, додавати легенди. Покажемо, як візуалізувати дані зі списків та файлів.

1. Вступ до Turtle: перші кроки в графіці

Модуль turtle – це одна з найстаріших і найулюбленіших бібліотек Python для навчання програмуванню. Вона реалізує класичну "черепащачу графіку", де маленька черепашка з пензликом у хвості рухається по екрану і залишає слід.

Чому варто почати з Turtle?

- **Візуальна зворотна зв'язка** – ви одразу бачите результат своїх команд
- **Інтуїтивність** – команди схожі на те, що ви б сказали черепащі в реальності
- **Весело** – можна створювати красиві малюнки кількома рядками коду
- **Основи алгоритмів** – чудово ілюструє роботу циклів та функцій

Перша програма з Turtle

```
import turtle
```

Створюємо вікно та черепашку

```
screen = turtle.Screen()
screen.title("Моя перша черепашка")
screen.bgcolor("lightblue")
```

Створюємо черепашку

```
t = turtle.Turtle()
t.shape("turtle")
t.color("green")
t.pensize(3)
```

Малюємо квадрат

```
t.forward(100) # вперед 100 пікселів
t.left(90)    # поворот ліворуч на 90 градусів
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
```

Чекаємо на клік для закриття

```
screen.exitonclick()
```

Основні команди Turtle

Команда	Опис
forward(distance)	Рухається вперед на distance пікселів
backward(distance)	Рухається назад
right(angle)	Повертає праворуч на angle градусів
left(angle)	Повертає ліворуч на angle градусів
goto(x, y)	Переміщується в точку (x, y)
setx(x)	Встановлює x-координату

Команда	Опис
sety(y)	Встановлює у-координату
home()	Повертається в центр (0, 0)
circle(radius)	Малює коло заданого радіуса
dot(size)	Малює точку
penup()	Піднімає перо (не малює при русі)
pendown()	Опускає перо
pensize(width)	Встановлює товщину лінії
pencolor(color)	Встановлює колір пера
fillcolor(color)	Встановлює колір заливки
begin_fill()	Починає заливку
end_fill()	Завершує заливку
speed(speed)	Встановлює швидкість (1-10, 0 - найшвидше)
hideturtle()	Ховає черепашку
showturtle()	Показує черепашку
clear()	Очищає малюнок
reset()	Скидає все до початкового стану

Прості приклади

```
import turtle
```

```
def draw_star():
    """Малює зірку"""
```



```

screen = turtle.Screen()
screen.bgcolor("black")
screen.title("Зірка")

t = turtle.Turtle()
t.color("yellow")
t.pensize(2)
t.speed(5)

for i in range(5):
    t.forward(200)
    t.right(144) # 180 - 36 = 144 градуси для зірки

screen.exitonclick()

def draw_flower():
    """Малює квітку"""
    screen = turtle.Screen()
    screen.bgcolor("white")
    screen.title("Квітка")

    t = turtle.Turtle()
    t.color("red")
    t.pensize(2)
    t.speed(8)

    for i in range(36):
        for j in range(4):
            t.forward(100)
            t.right(90)
            t.right(10)

    screen.exitonclick()

def draw_spiral():
    """Малює спіраль"""
    screen = turtle.Screen()

```

```
screen.bgcolor("white")
screen.title("Спіраль")
```

```
t = turtle.Turtle()
t.color("blue")
t.speed(10)
```

```
for i in range(100):
    t.forward(i * 2)
    t.right(90)
```

```
screen.exitonclick()
```

```
# Розкоментуйте для виконання
# draw_star()
# draw_flower()
# draw_spiral()
```

Налаштування кольорів

```
import turtle
```

```
def color_demo():
    """Демонстрація роботи з кольорами"""
    screen = turtle.Screen()
    screen.bgcolor("black")
    screen.title("Кольори")
```

```
t = turtle.Turtle()
t.speed(5)
t.pensize(3)
```

```
# Малюємо різнокольорові лінії
colors = ["red", "orange", "yellow", "green", "blue", "purple", "pink"]
```

```
for i, color in enumerate(colors):
    t.pencolor(color)
    t.forward(200)
```

```
t.right(90)
t.forward(20)
t.right(90)
t.forward(200)
t.left(90)
t.forward(20)
t.left(90)
```

```
# Малюємо кольорові точки
```

```
t.penup()
t.goto(-150, -150)
```

```
for i in range(10):
    t.pendown()
    t.dot(20, colors[i % len(colors)])
    t.penup()
    t.forward(30)
```

```
screen.exitonclick()
```

```
# color_demo()
```

Керування пером та позиціонування

```
import turtle
```

```
def pen_control():
    """Демонстрація керування пером"""
    screen = turtle.Screen()
    screen.bgcolor("white")
    screen.title("Керування пером")
```

```
t = turtle.Turtle()
t.speed(3)
t.pensize(2)
```

```
# Малюємо пунктирну лінію
```

```
for i in range(10):
```

```

t.pendown()
t.forward(20)
t.penup()
t.forward(10)

# Переміщуємося без малювання
t.penup()
t.goto(-100, 100)
t.pendown()

# Малюємо іншу лінію
t.pencolor("red")
t.forward(200)

# Переміщуємо абсолютно
t.penup()
t.goto(0, 0)
t.pendown()
t.dot(50, "blue")

screen.exitonclick()

# pen_control()

```

2. Малювання геометричних фігур та патернів за допомогою циклів

Цикли – це ідеальний інструмент для малювання повторюваних патернів. Turtle у поєднанні з циклами дозволяє створювати вражаючі малюнки з мінімальним кодом.

Малювання квадрата за допомогою циклу

```

import turtle

def draw_square(size):
    """Малює квадрат заданого розміру"""
    for _ in range(4):

```

```
turtle.forward(size)
turtle.right(90)
```

```
def squares_demo():
    """Демонстрація малювання квадратів"""
    screen = turtle.Screen()
    screen.bgcolor("white")
    screen.title("Квадрати")

    t = turtle.Turtle()
    t.speed(5)
    t.pensize(2)

    # Малюємо кілька квадратів різного розміру
    colors = ["red", "blue", "green", "orange", "purple"]
    sizes = [50, 100, 150, 200, 250]

    for i in range(5):
        t.pencolor(colors[i])
        t.penup()
        t.goto(-sizes[i]//2, -sizes[i]//2) # центруємо
        t.pendown()
        draw_square(sizes[i])

    screen.exitonclick()

# squares_demo()
```

Малювання правильних багатокутників

```
import turtle

def draw_polygon(sides, side_length):
    """
    Малює правильний багатокутник.
    sides - кількість сторін
    side_length - довжина сторони
    """
```

```

angle = 360 / sides
for _ in range(sides):
    turtle.forward(side_length)
    turtle.right(angle)

def polygons_demo():
    """Демонстрація різних багатокутників"""
    screen = turtle.Screen()
    screen.bgcolor("black")
    screen.title("Многокутники")

    t = turtle.Turtle()
    t.speed(5)
    t.pensize(2)

    polygons = [
        (3, "red", "Трикутник"),
        (4, "orange", "Квадрат"),
        (5, "yellow", "П'ятикутник"),
        (6, "green", "Шестикутник"),
        (8, "blue", "Восьмикутник"),
        (12, "purple", "Дванадцятикутник")
    ]

    start_x = -250
    for i, (sides, color, name) in enumerate(polygons):
        t.penup()
        t.goto(start_x + i * 100, 0)
        t.pendown()
        t.pencolor(color)
        t.write(name, font=("Arial", 10, "normal"))
        t.penup()
        t.goto(start_x + i * 100, -20)
        t.pendown()
        draw_polygon(sides, 50)

    screen.exitonclick()

```

```
# polygons_demo()
```

Вкладені цикли: створення патернів

```
import turtle
```

```
def pattern_demo():
```

```
    """Малювання складних патернів за допомогою вкладених циклів"""
```

```
    screen = turtle.Screen()
```

```
    screen.bgcolor("white")
```

```
    screen.title("Патерни")
```

```
    t = turtle.Turtle()
```

```
    t.speed(10)
```

```
    t.pensize(1)
```

```
    # Патерн 1: Квітка з квадратів
```

```
    t.penup()
```

```
    t.goto(-300, 150)
```

```
    t.pendown()
```

```
    t.pencolor("red")
```

```
    for i in range(36):
```

```
        for j in range(4):
```

```
            t.forward(50)
```

```
            t.right(90)
```

```
        t.right(10)
```

```
    # Патерн 2: Спіраль з кольорами
```

```
    t.penup()
```

```
    t.goto(150, 150)
```

```
    t.pendown()
```

```
    colors = ["red", "orange", "yellow", "green", "blue", "purple"]
```

```
    length = 10
```

```
    for i in range(100):
```

```
t.pencolor(colors[i % len(colors)])
t.forward(length)
t.right(89)
length += 2
```

Патерн 3: Сітка з точок

```
t.penup()
t.goto(-300, -200)
t.pendown()

t.pencolor("black")
for i in range(10):
    for j in range(10):
        t.dot(5)
        t.penup()
        t.forward(20)
        t.pendown()
    t.penup()
    t.goto(-300, -200 + (i + 1) * 20)
    t.pendown()
```

```
screen.exitonclick()
```

pattern_demo()

Малювання з заповненням кольором

```
import turtle

def fill_demo():
    """Демонстрація заповнення кольором"""
    screen = turtle.Screen()
    screen.bgcolor("lightgray")
    screen.title("Заливка кольором")

    t = turtle.Turtle()
    t.speed(5)
```



```
# Малюємо трикутник із заливкою
t.penup()
t.goto(-200, 0)
t.pendown()

t.fillcolor("yellow")
t.begin_fill()
for _ in range(3):
    t.forward(150)
    t.left(120)
t.end_fill()

# Малюємо квадрат із заливкою
t.penup()
t.goto(50, 0)
t.pendown()

t.fillcolor("lightblue")
t.begin_fill()
for _ in range(4):
    t.forward(150)
    t.right(90)
t.end_fill()

# Малюємо коло із заливкою
t.penup()
t.goto(-75, -150)
t.pendown()

t.fillcolor("lightgreen")
t.begin_fill()
t.circle(75)
t.end_fill()

screen.exitonclick()

# fill_demo()
```

Практичний приклад: малювання сонячної системи (спрощено)

```
import turtle
import math

def draw_planet(t, radius, color, distance, speed):
    """Малює планету на орбіті"""
    t.penup()
    t.goto(distance, 0)
    t.pendown()
    t.dot(radius * 2, color)
    t.penup()

def solar_system():
    """Малює спрощену модель сонячної системи"""
    screen = turtle.Screen()
    screen.bgcolor("black")
    screen.title("Сонячна система")
    screen.setup(800, 600)

    t = turtle.Turtle()
    t.speed(0)
    t.hideturtle()

    # Сонце
    t.penup()
    t.goto(0, -50)
    t.pendown()
    t.fillcolor("yellow")
    t.begin_fill()
    t.circle(50)
    t.end_fill()
    t.penup()
    t.goto(0, 0)
    t.color("yellow")
    t.write("Сонце", align="center", font=("Arial", 12, "bold"))
```

```
# Орбіти планет
```

```
t.penup()
t.goto(0, 0)
t.pendown()
t.pencolor("gray")

distances = [120, 180, 240, 300, 360]
for d in distances:
    t.circle(d)
    t.penup()
    t.goto(0, 0)
    t.pendown()
```

```
# Планети
```

```
planets = [
    (120, "gray", "Меркурій"),
    (180, "orange", "Венера"),
    (240, "blue", "Земля"),
    (300, "red", "Марс"),
    (360, "brown", "Юпітер")
]

for distance, color, name in planets:
    t.penup()
    t.goto(distance, 0)
    t.pendown()
    t.dot(15, color)
    t.penup()
    t.goto(distance, -20)
    t.color("white")
    t.write(name, align="center", font=("Arial", 8, "normal"))

screen.exitonclick()
```

```
# solar_system()
```

Практичний приклад: малювання дерева за допомогою рекурсії (ознайомлення)

```

import turtle

def draw_tree(branch_len, t):
    """
    Малює дерево рекурсивно.
    branch_len - довжина поточної гілки
    t - об'єкт черепашки
    """
    if branch_len < 5: # базовий випадок
        return

    # Малюємо основну гілку
    t.forward(branch_len)

    # Малюємо праву гілку
    t.right(20)
    draw_tree(branch_len - 15, t)

    # Повертаємося
    t.left(40)
    draw_tree(branch_len - 15, t)

    # Повертаємося в початкове положення
    t.right(20)
    t.backward(branch_len)

def tree_demo():
    """Демонстрація малювання дерева"""
    screen = turtle.Screen()
    screen.bgcolor("lightblue")
    screen.title("Рекурсивне дерево")

    t = turtle.Turtle()
    t.speed(0)
    t.pensize(2)
    t.color("brown")

```

```

# Початкове позиціонування
t.penup()
t.goto(0, -200)
t.pendown()
t.left(90)

# Малюємо дерево
draw_tree(100, t)

t.hideturtle()
screen.exitonclick()

# tree_demo()

```

Практичний приклад: малювання сніжинки Коха (фрактал)

```

import turtle

def koch_curve(t, length, depth):
    """
    Малює криву Коха (сніжинку) рекурсивно.
    length - довжина лінії
    depth - глибина рекурсії
    """
    if depth == 0:
        t.forward(length)
    else:
        length /= 3.0
        koch_curve(t, length, depth-1)
        t.left(60)
        koch_curve(t, length, depth-1)
        t.right(120)
        koch_curve(t, length, depth-1)
        t.left(60)
        koch_curve(t, length, depth-1)

def snowflake_demo():

```

```

"""Малює сніжинку Коха"""
screen = turtle.Screen()
screen.bgcolor("black")
screen.title("Сніжинка Коха")
screen.setup(800, 800)

t = turtle.Turtle()
t.speed(0)
t.penup()
t.goto(-200, 100)
t.pendown()
t.pencolor("white")
t.pensize(2)

# Малюємо три сторони сніжинки
for _ in range(3):
    koch_curve(t, 400, 3)
    t.right(120)

t.hideturtle()
screen.exitonclick()

# snowflake_demo()

```

Інтерактивне малювання

```

import turtle

def interactive_drawing():
    """Демонстрація інтерактивного малювання з клавіатурою"""
    screen = turtle.Screen()
    screen.title("Інтерактивне малювання")
    screen.bgcolor("white")
    screen.setup(600, 600)

    t = turtle.Turtle()
    t.speed(3)
    t.pensize(2)

```

Функції для керування

```
def move_forward():
```

```
    t.forward(20)
```

```
def move_backward():
```

```
    t.backward(20)
```

```
def turn_left():
```

```
    t.left(15)
```

```
def turn_right():
```

```
    t.right(15)
```

```
def pen_up():
```

```
    t.penup()
```

```
def pen_down():
```

```
    t.pendown()
```

```
def clear_screen():
```

```
    t.clear()
```

```
def change_color_red():
```

```
    t.pencolor("red")
```

```
def change_color_blue():
```

```
    t.pencolor("blue")
```

```
def change_color_green():
```

```
    t.pencolor("green")
```

Прив'язка клавіш

```
screen.listen()
```

```
screen.onkey(move_forward, "Up")
```

```
screen.onkey(move_backward, "Down")
```

```
screen.onkey(turn_left, "Left")
```

```
screen.onkey(turn_right, "Right")
screen.onkey(pen_up, "p")
screen.onkey(pen_down, "d")
screen.onkey(clear_screen, "c")
screen.onkey(change_color_red, "r")
screen.onkey(change_color_blue, "b")
screen.onkey(change_color_green, "g")
```

```
# Інструкції
```

```
t.penup()
t.goto(0, 250)
t.write("Стрілки - рух, Р - перо вгору, Д - перо вниз",
       align="center", font=("Arial", 12, "normal"))
t.goto(0, 230)
t.write("С - очистити, R/B/G - колір",
       align="center", font=("Arial", 12, "normal"))
t.goto(0, 0)
t.pendown()
```

```
screen.exitonclick()
```

```
# interactive_drawing()
```

Збереження малюнка

```
import turtle
```

```
def save_drawing():
```

```
    """Малює і зберігає результат"""
```

```
    screen = turtle.Screen()
```

```
    screen.bgcolor("white")
```

```
    screen.title("Збереження малюнка")
```

```
    t = turtle.Turtle()
```

```
    t.speed(5)
```

```
# Малюємо щось красиве
```

```
colors = ["red", "orange", "yellow", "green", "blue", "purple"]
```



```

for i in range(36):
    t.pencolor(colors[i % len(colors)])
    t.circle(100)
    t.right(10)

# Зберігаємо малюнок
canvas = screen.getcanvas()
canvas.postscript(file="drawing.eps")
print("✓ Малюнок збережено у файл drawing.eps")

screen.exitonclick()

# save_drawing()

```

Підсумки

- **Turtle** – чудовий інструмент для візуального навчання програмуванню
- **Цикли** дозволяють створювати складні патерни з мінімальним кодом
- **Рекурсія** ідеально підходить для малювання фракталів
- **Кольори та заливка** роблять малюнки яскравішими
- **Інтерактивність** дозволяє керувати малюванням у реальному часі

У наступному пункті ми розглянемо систему координат, фрактали та рекурсію в Turtle більш детально, а потім перейдемо до професійної візуалізації даних з Matplotlib.

3. Система координат, фрактали та рекурсія в Turtle

Розуміння системи координат у Turtle відкриває нові можливості для точного позиціонування об'єктів. А поєднання з рекурсією дозволяє створювати вражаючі фрактальні зображення.

Система координат у Turtle

Вікно Turtle має стандартну декартову систему координат:

- Центр екрану – точка (0, 0)
- Вісь X: додатні значення праворуч, від'ємні ліворуч
- Вісь Y: додатні значення вгору, від'ємні вниз

```

import turtle

def coordinate_demo():
    """Демонстрація системи координат"""
    screen = turtle.Screen()
    screen.bgcolor("white")
    screen.title("Система координат")
    screen.setup(600, 600)

    t = turtle.Turtle()
    t.speed(3)
    t.pensize(2)

    # Малюємо осі координат
    t.penup()
    t.goto(-250, 0)
    t.pendown()
    t.forward(500)
    t.penup()
    t.goto(0, -250)
    t.pendown()
    t.left(90)
    t.forward(500)

    # Позначаємо центр
    t.penup()
    t.goto(0, 0)
    t.dot(10, "red")
    t.write("(0, 0)", align="left", font=("Arial", 10, "normal"))

    # Малюємо точки з різними координатами
    points = [
        (100, 100, "blue", "A(100, 100)"),
        (-100, 100, "green", "B(-100, 100)"),
        (-100, -100, "orange", "C(-100, -100)"),
        (100, -100, "purple", "D(100, -100)")
    ]

```

```
for x, y, color, label in points:
    t.penup()
    t.goto(x, y)
    t.dot(10, color)
    t.write(label, align="left", font=("Arial", 8, "normal"))
```

```
screen.exitonclick()
```

```
# coordinate_demo()
```

Абсолютне позиціонування

```
import turtle
```

```
def absolute_positioning():
    """Демонстрація абсолютного позиціонування"""
    screen = turtle.Screen()
    screen.bgcolor("lightblue")
    screen.title("Абсолютне позиціонування")
    screen.setup(800, 600)

    t = turtle.Turtle()
    t.speed(3)
    t.pensize(2)

    # Малюємо будинок з використанням goto
    t.penup()
    t.goto(-150, -100)
    t.pendown()

    # Стіни
    t.fillcolor("yellow")
    t.begin_fill()
    for _ in range(4):
        t.forward(300)
        t.left(90)
    t.end_fill()
```

Дак

```
t.penup()
t.goto(-150, 200)
t.pendown()
t.fillcolor("red")
t.begin_fill()
t.goto(0, 300)
t.goto(150, 200)
t.goto(-150, 200)
t.end_fill()
```

Двери

```
t.penup()
t.goto(25, -100)
t.pendown()
t.fillcolor("brown")
t.begin_fill()
for _ in range(2):
    t.forward(50)
    t.left(90)
    t.forward(100)
    t.left(90)
t.end_fill()
```

Вікна

```
windows = [(-100, 50), (50, 50)]
for x, y in windows:
    t.penup()
    t.goto(x, y)
    t.pendown()
    t.fillcolor("lightblue")
    t.begin_fill()
    for _ in range(4):
        t.forward(50)
        t.left(90)
    t.end_fill()
```

```

# Рамки
t.penup()
t.goto(x + 25, y)
t.pendown()
t.forward(0)
t.penup()
t.goto(x, y + 25)
t.pendown()
t.left(90)
t.forward(50)

screen.exitonclick()

```

absolute_positioning()

Отримання поточної позиції

```
import turtle
```

```

def track_position():
    """Відстеження позиції черепашки"""
    screen = turtle.Screen()
    screen.bgcolor("white")
    screen.title("Відстеження позиції")

    t = turtle.Turtle()
    t.speed(1)

    # Малюємо лінію з виведенням координат
    for i in range(5):
        t.forward(100)
        print(f"Позиція: ({t.xcor():.0f}, {t.ycor():.0f})")
        t.right(90)

    # Показуємо поточну позицію на екрані
    t.penup()
    t.goto(-50, 150)

```

```
t.write(f"Поточна позиція: ({t.xcor():.0f}, {t.ycor():.0f})",
       font=("Arial", 12, "normal"))

screen.exitonclick()
```

track_position()

Фрактали: рекурсивне дерево

Фрактали – це геометричні фігури, які самоподібні: кожна частина схожа на ціле. Рекурсія ідеально підходить для їх малювання.

```
import turtle
import random

def recursive_tree(branch_len, t, angle=20, reduction=0.7):
    """
    Малює рекурсивне дерево.
    branch_len - довжина поточної гілки
    t - об'єкт черепашки
    angle - кут розгалуження
    reduction - коефіцієнт зменшення довжини гілок
    """
    if branch_len < 10: # базовий випадок
        t.color("green")
        t.dot(5)
        t.color("brown")
        return

    # Встановлюємо товщину лінії пропорційно довжині гілки
    t.pensize(branch_len // 10)

    # Малюємо основну гілку
    t.forward(branch_len)

    # Малюємо праву гілку
    t.right(angle)
    recursive_tree(branch_len * reduction, t, angle, reduction)
```

```

# Малюємо ліву гілку
t.left(angle * 2)
recursive_tree(branch_len * reduction, t, angle, reduction)

# Повертаємося назад
t.right(angle)
t.backward(branch_len)

def tree_demo_advanced():
    """Покращена версія рекурсивного дерева"""
    screen = turtle.Screen()
    screen.bgcolor("skyblue")
    screen.title("Рекурсивне дерево")
    screen.setup(800, 600)

    t = turtle.Turtle()
    t.speed(0)
    t.left(90)
    t.penup()
    t.goto(0, -200)
    t.pendown()
    t.color("brown")

    # Параметри дерева
    recursive_tree(100, t, angle=25, reduction=0.7)

    t.hideturtle()
    screen.exitonclick()

# tree_demo_advanced()

```

Сніжинка Коха (класичний фрактал)

```

import turtle

def koch_curve(t, length, depth):
    """
    Малює криву Коха.
    """

```

```

length - довжина лінії
depth - глибина рекурсії
"""

if depth == 0:
    t.forward(length)
else:
    length /= 3.0
    koch_curve(t, length, depth-1)
    t.left(60)
    koch_curve(t, length, depth-1)
    t.right(120)
    koch_curve(t, length, depth-1)
    t.left(60)
    koch_curve(t, length, depth-1)

def koch_snowflake(t, length, depth):
    """Малює сніжинку Коха (три криві)"""
    for _ in range(3):
        koch_curve(t, length, depth)
        t.right(120)

def snowflake_animation():
    """Анімація сніжинки з різною глибиною"""
    screen = turtle.Screen()
    screen.bgcolor("black")
    screen.title("Сніжинка Коха")
    screen.setup(800, 800)

    t = turtle.Turtle()
    t.speed(0)
    t.pencolor("white")

    # Малюємо сніжинки з різною глибиною
    positions = [(-300, 150), (100, 150), (-300, -200), (100, -200)]
    depths = [0, 1, 2, 3]

    for i, (x, y) in enumerate(positions):

```



```

t.penup()
t.goto(x, y)
t.pendown()
t.clear()
t.write(f"Глибина {depths[i]}", font=("Arial", 12, "normal"))
t.penup()
t.goto(x + 150, y - 50)
t.pendown()
koch_snowflake(t, 200, depths[i])

```

```
screen.exitonclick()
```

```
# snowflake_animation()
```

Фрактальний трикутник Серпінського

```
import turtle
```

```
def sierpinski(t, points, depth):
```

```
    """
```

Малює трикутник Серпінського.

points - список з трьох точок (кортежів (x, y))

depth - глибина рекурсії

```
    """
```

```
def draw_triangle(p1, p2, p3):
```

```
    t.penup()
```

```
    t.goto(p1)
```

```
    t.pendown()
```

```
    t.goto(p2)
```

```
    t.goto(p3)
```

```
    t.goto(p1)
```

```
def mid(p1, p2):
```

```
    """Повертає середину між двома точками"""
```

```
    return ((p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2)
```

```
if depth == 0:
```

```
    draw_triangle(points[0], points[1], points[2])
```

else:

Знаходимо середини сторін

m1 = mid(points[0], points[1])

m2 = mid(points[1], points[2])

m3 = mid(points[2], points[0])

Рекурсивно малюємо три менші трикутники

sierpinski(t, [points[0], m1, m3], depth-1)

sierpinski(t, [m1, points[1], m2], depth-1)

sierpinski(t, [m3, m2, points[2]], depth-1)

def sierpinski_demo():

"""Демонстрація трикутника Серпінського"""

screen = turtle.Screen()

screen.bgcolor("black")

screen.title("Трикутник Серпінського")

screen.setup(800, 800)

t = turtle.Turtle()

t.speed(0)

t.pencolor("yellow")

Вершини великого трикутника

points = [(-300, -200), (300, -200), (0, 300)]

sierpinski(t, points, 5)

t.hideturtle()

screen.exitonclick()

sierpinski_demo()

Фрактал "Папороть"

import turtle

import random

def fern_fractal(t, x, y, size, angle):

```
"""
```

Малює фрактал папороті (система ітерованих функцій - спрощено)

```
"""
```

```
if size < 2:
```

```
    t.goto(x, y)
```

```
    t.dot(2, "green")
```

```
    return
```

```
# Основне стебло
```

```
t.penup()
```

```
t.goto(x, y)
```

```
t.pendown()
```

```
t.setheading(angle)
```

```
t.forward(size)
```

```
new_x, new_y = t.xcor(), t.ycor()
```

```
# Ліві гілки
```

```
t.setheading(angle + 30)
```

```
fern_fractal(t, new_x, new_y, size * 0.6, angle + 30)
```

```
# Праві гілки
```

```
t.setheading(angle - 30)
```

```
fern_fractal(t, new_x, new_y, size * 0.6, angle - 30)
```

```
# Маленька центральна гілка
```

```
t.setheading(angle)
```

```
fern_fractal(t, new_x, new_y, size * 0.3, angle)
```

```
def fern_demo():
```

```
    """Демонстрація фрактальної папороті"""
```

```
    screen = turtle.Screen()
```

```
    screen.bgcolor("black")
```

```
    screen.title("Фрактальна папороть")
```

```
    screen.setup(800, 800)
```

```
t = turtle.Turtle()
```

```
t.speed(0)
```

```

t.pencolor("lightgreen")

fern_fractal(t, 0, -300, 100, 90)

t.hideturtle()
screen.exitonclick()

# fern_demo()

```

4. Вступ до Matplotlib: перша візуалізація даних

Matplotlib – це найпопулярніша бібліотека Python для створення графіків та візуалізації даних. Вона надзвичайно потужна і гнучка, дозволяючи створювати графіки будь-якої складності.

Встановлення та імпорт

```

# Встановлення (якщо не встановлено)
# pip install matplotlib

import matplotlib.pyplot as plt
import numpy as np # для створення масивів даних

```

Перший простий графік

```

import matplotlib.pyplot as plt

def first_plot():
    """Створення першого простого графіка"""

    # Дані для графіка
    x = [1, 2, 3, 4, 5]
    y = [2, 4, 6, 8, 10]

    # Створення графіка
    plt.plot(x, y)

```

```

# Додавання заголовка та підписів осей
plt.title("Мій перший графік")
plt.xlabel("X вісь")
plt.ylabel("Y вісь")

# Відображення графіка
plt.show()

# first_plot()

```

Кілька графіків на одній площині

```

def multiple_lines():
    """Кілька ліній на одному графіку"""

    x = [1, 2, 3, 4, 5]
    y1 = [1, 4, 9, 16, 25]
    y2 = [1, 2, 3, 4, 5]
    y3 = [25, 16, 9, 4, 1]

    plt.plot(x, y1, label='y = x²', color='red', linewidth=2)
    plt.plot(x, y2, label='y = x', color='blue', linewidth=2, linestyle='--')
    plt.plot(x, y3, label='y = 26 - x²', color='green', linewidth=2, marker='o')

    plt.title("Кілька графіків")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend() # показує легенду
    plt.grid(True, alpha=0.3) # сітка

    plt.show()

# multiple_lines()

```

Різні стилі ліній та маркери

```

def styles_demo():
    """Демонстрація різних стилів ліній та маркерів"""

```

```

x = [1, 2, 3, 4, 5]

# Різні стилі
styles = [
    ('-', 'solid', 'blue'),
    ('--', 'dashed', 'red'),
    ('-.', 'dashdot', 'green'),
    (':', 'dotted', 'orange')
]

for i, (linestyle, label, color) in enumerate(styles):
    y = [1, 2 + i, 3 + i, 4 + i, 5 + i]
    plt.plot(x, y, linestyle=linestyle, color=color,
             label=label, linewidth=2, marker='o')

plt.title("Стилі ліній")
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# styles_demo()

```

Точкові діаграми (scatter plot)

```

def scatter_demo():
    """Точкова діаграма (розкиду)"""

    # Генеруємо випадкові дані
    np.random.seed(42)
    x = np.random.rand(50) * 10
    y = np.random.rand(50) * 10
    colors = np.random.rand(50)
    sizes = np.random.rand(50) * 500

    plt.scatter(x, y, c=colors, s=sizes, alpha=0.6, cmap='viridis')
    plt.colorbar() # показує шкалу кольорів

```

```
plt.title("Точкова діаграма")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True, alpha=0.3)

plt.show()
```

```
# scatter_demo()
```

Стовпчасті діаграми (bar chart)

```
def bar_chart_demo():
    """Стовпчаста діаграма"""

    # Дані про продажі
    categories = ['Продукт А', 'Продукт В', 'Продукт С', 'Продукт D',
'Продукт E']
    sales = [150, 230, 180, 275, 190]
    colors = ['red', 'blue', 'green', 'orange', 'purple']

    plt.bar(categories, sales, color=colors, alpha=0.7)

    plt.title("Продажі продуктів")
    plt.xlabel("Продукти")
    plt.ylabel("Кількість продажів")

    # Додаємо значення над стовпчиками
    for i, v in enumerate(sales):
        plt.text(i, v + 5, str(v), ha='center', va='bottom')

    plt.grid(True, alpha=0.3, axis='y')
    plt.show()

# bar_chart_demo()
```

Горизонтальна стовпчаста діаграма

```
def barh_demo():
```

```
"""Горизонтальна стовпчаста діаграма"""
```

```
categories = ['Дуже задоволені', 'Задоволені', 'Нейтрально',  
'Незадоволені', 'Дуже незадоволені']  
percentages = [45, 30, 15, 7, 3]  
colors = ['green', 'lightgreen', 'yellow', 'orange', 'red']  
  
plt.barh(categories, percentages, color=colors, alpha=0.7)  
  
plt.title("Рівень задоволеності клієнтів")  
plt.xlabel("Відсоток (%)")  
  
for i, v in enumerate(percentages):  
    plt.text(v + 1, i, f'{v}%', va='center')  
  
plt.show()  
  
# barh_demo()
```

Кругові діаграми (pie chart)

```
def pie_chart_demo():  
    """Кругова діаграма"""  
  
    # Дані про частки ринку  
    companies = ['Компанія А', 'Компанія В', 'Компанія С', 'Інші']  
    market_share = [35, 25, 20, 20]  
    colors = ['#ff9999', '#66b3ff', '#99ff99', '#ffcc99']  
  
    # Відокремлюємо найбільший сектор  
    explode = (0.1, 0, 0, 0)  
  
    plt.pie(market_share, labels=companies, colors=colors,  
            autopct='%1.1f%%', startangle=90, explode=explode,  
            shadow=True)  
  
    plt.title("Частка ринку")  
    plt.axis('equal') # робимо коло круглим
```



```
plt.show()
```

```
# pie_chart_demo()
```

Гістограми (histogram)

```
def histogram_demo():
```

```
    """Гістограма розподілу даних"""
```

```
    # Генеруємо нормальний розподіл
```

```
    np.random.seed(42)
```

```
    data = np.random.randn(1000) * 10 + 50 # середнє 50, std 10
```

```
    plt.hist(data, bins=30, edgecolor='black', alpha=0.7, color='blue')
```

```
    plt.title("Розподіл віку клієнтів")
```

```
    plt.xlabel("Вік")
```

```
    plt.ylabel("Частота")
```

```
    plt.grid(True, alpha=0.3)
```

```
    # Додаємо лінію середнього
```

```
    mean_value = np.mean(data)
```

```
    plt.axvline(mean_value, color='red', linestyle='--',
```

```
                label=f'Середнє: {mean_value:.1f}')
```

```
    plt.legend()
```

```
    plt.show()
```

```
# histogram_demo()
```

Кілька підграфіків (subplots)

```
def subplots_demo():
```

```
    """Демонстрація кількох підграфіків"""
```

```
    fig, axes = plt.subplots(2, 2, figsize=(10, 8))
```

```
    fig.suptitle("Різні типи графіків", fontsize=16)
```

```
x = np.linspace(0, 10, 100)
```

```
# Графік 1: синус
```

```
axes[0, 0].plot(x, np.sin(x), 'r-')
```

```
axes[0, 0].set_title('sin(x)')
```

```
axes[0, 0].grid(True)
```

```
# Графік 2: косинус
```

```
axes[0, 1].plot(x, np.cos(x), 'b-')
```

```
axes[0, 1].set_title('cos(x)')
```

```
axes[0, 1].grid(True)
```

```
# Графік 3: exp(-x) * sin(x)
```

```
axes[1, 0].plot(x, np.exp(-x) * np.sin(x), 'g-')
```

```
axes[1, 0].set_title('exp(-x) * sin(x)')
```

```
axes[1, 0].grid(True)
```

```
# Графік 4: x2
```

```
axes[1, 1].plot(x, x**2, 'purple')
```

```
axes[1, 1].set_title('x2')
```

```
axes[1, 1].grid(True)
```

```
plt.tight_layout()
```

```
plt.show()
```

```
# subplots_demo()
```

Практичний приклад: візуалізація даних про студентів

```
def students_visualization():
```

```
    """Візуалізація даних про студентів"""
```

```
# Дані про студентів
```

```
students = ['Олена', 'Іван', 'Марія', 'Петро', 'Анна']
```

```
grades = {
```

```
    'Математика': [85, 90, 78, 65, 95],
```

```
    'Фізика': [78, 88, 82, 70, 92],
```

```
    'Програмування': [92, 85, 95, 60, 98],
```

```
    'Англійська': [88, 75, 90, 80, 85]
}
```

```
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
fig.suptitle("Аналіз успішності студентів", fontsize=16)
```

1. Стовпчаста діаграма: середній бал кожного студента

```
avg_grades = [sum(g) / len(g) for g in zip(*grades.values())]
axes[0, 0].bar(students, avg_grades, color='skyblue', edgecolor='black')
axes[0, 0].set_title('Середній бал студентів')
axes[0, 0].set_ylabel('Середній бал')
axes[0, 0].grid(True, alpha=0.3, axis='y')
```

Додаємо значення

```
for i, v in enumerate(avg_grades):
    axes[0, 0].text(i, v + 1, f'{v:.1f}', ha='center')
```

2. Точкова діаграма: математика vs програмування

```
axes[0, 1].scatter(grades['Математика'], grades['Програмування'],
                  c='red', s=100, alpha=0.6)
axes[0, 1].set_title('Математика vs Програмування')
axes[0, 1].set_xlabel('Математика')
axes[0, 1].set_ylabel('Програмування')
axes[0, 1].grid(True, alpha=0.3)
```

Додаємо підписи точок

```
for i, student in enumerate(students):
    axes[0, 1].annotate(student,
                        (grades['Математика'][i], grades['Програмування'][i]),
                        xytext=(5, 5), textcoords='offset points')
```

3. Гістограма розподілу оцінок

```
all_grades = [g for subject in grades.values() for g in subject]
axes[1, 0].hist(all_grades, bins=10, edgecolor='black',
               alpha=0.7, color='green')
axes[1, 0].set_title('Розподіл оцінок')
axes[1, 0].set_xlabel('Оцінка')
```

```
axes[1, 0].set_ylabel('Частота')
axes[1, 0].grid(True, alpha=0.3)
```

```
# 4. Кругова діаграма: відсоток відмінників
```

```
excellent = sum(1 for g in all_grades if g >= 90)
good = sum(1 for g in all_grades if 75 <= g < 90)
satisfactory = sum(1 for g in all_grades if 60 <= g < 75)
poor = sum(1 for g in all_grades if g < 60)
```

```
sizes = [excellent, good, satisfactory, poor]
labels = ['Відмінно (90+)', 'Добре (75-89)', 'Задовільно (60-74)',
'Незадовільно (<60)']
colors = ['#4CAF50', '#2196F3', '#FFC107', '#F44336']
```

```
axes[1, 1].pie(sizes, labels=labels, colors=colors,
               autopct='%1.1f%%', startangle=90)
axes[1, 1].set_title('Розподіл оцінок за категоріями')
```

```
plt.tight_layout()
plt.show()
```

```
# students_visualization()
```

Візуалізація з файлів

```
import csv
from datetime import datetime
```

```
def plot_from_file(filename):
    """Читає дані з CSV файлу та будує графік"""
```

```
dates = []
temperatures = []
```

```
# Читаємо дані з файлу
```

```
with open(filename, 'r', encoding='utf-8') as f:
    reader = csv.reader(f)
    next(reader) # пропускаємо заголовок
```

```

for row in reader:
    date = datetime.strptime(row[0], '%Y-%m-%d')
    temp = float(row[1])
    dates.append(date)
    temperatures.append(temp)

# Будуємо графік
plt.figure(figsize=(12, 6))
plt.plot(dates, temperatures, 'b-', linewidth=2, marker='o', markersize=4)

plt.title("Температура повітря за рік", fontsize=16)
plt.xlabel("Дата")
plt.ylabel("Температура (°C)")
plt.grid(True, alpha=0.3)

# Додаємо лінію середньої температури
mean_temp = sum(temperatures) / len(temperatures)
plt.axhline(mean_temp, color='red', linestyle='--',
            label=f'Середня: {mean_temp:.1f}°C')

plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Створюємо тестові дані
def create_test_data():
    """Створює тестовий CSV файл з даними про температуру"""
    import random
    from datetime import datetime, timedelta

    with open('temperature.csv', 'w', newline="", encoding='utf-8') as f:
        writer = csv.writer(f)
        writer.writerow(['Date', 'Temperature'])

    start_date = datetime(2023, 1, 1)
    for i in range(365):

```

```

date = start_date + timedelta(days=i)
# Створюємо сезонні коливання
temp = 15 + 15 * np.sin(i * 2 * np.pi / 365) + random.gauss(0, 3)
writer.writerow([date.strftime('%Y-%m-%d'), round(temp, 1)])

# create_test_data()
# plot_from_file('temperature.csv')

```

Інтерактивні можливості

```

def interactive_features():
    """Демонстрація інтерактивних можливостей"""

    x = np.linspace(0, 10, 100)
    y1 = np.sin(x)
    y2 = np.cos(x)

    fig, ax = plt.subplots(figsize=(10, 6))

    line1, = ax.plot(x, y1, 'r-', label='sin(x)', linewidth=2)
    line2, = ax.plot(x, y2, 'b-', label='cos(x)', linewidth=2)

    ax.set_title("Інтерактивний графік (натисніть клавіші 1/2)")
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.legend()
    ax.grid(True, alpha=0.3)

    # Функція для обробки натискань клавіш
    def on_key(event):
        if event.key == '1':
            line1.set_visible(not line1.get_visible())
        elif event.key == '2':
            line2.set_visible(not line2.get_visible())
        elif event.key == 'g':
            ax.grid(not ax.grid())

    fig.canvas.draw()

```

```
fig.canvas.mpl_connect('key_press_event', on_key)
```

```
plt.show()
```

```
# interactive_features()
```

Збереження графіків

```
def save_plot():
```

```
    """Створення та збереження графіка"""
```

```
    x = [1, 2, 3, 4, 5]
```

```
    y = [10, 15, 13, 18, 20]
```

```
    plt.figure(figsize=(8, 5))
```

```
    plt.plot(x, y, 'bo-', linewidth=2, markersize=8)
```

```
    plt.title("Приклад графіка")
```

```
    plt.xlabel("X")
```

```
    plt.ylabel("Y")
```

```
    plt.grid(True, alpha=0.3)
```

```
# Збереження в різних форматах
```

```
    plt.savefig('my_plot.png', dpi=300, bbox_inches='tight')
```

```
    plt.savefig('my_plot.pdf', bbox_inches='tight')
```

```
    plt.savefig('my_plot.svg', bbox_inches='tight')
```

```
    print("✓ Графіки збережено")
```

```
    plt.show()
```

```
# save_plot()
```

Підсумки

- **Система координат** в Turtle дозволяє точно позиціонувати об'єкти
- **Рекурсія** ідеально підходить для малювання фракталів
- **Matplotlib** – потужна бібліотека для професійної візуалізації даних
- Різні типи графіків (лінійні, стовпчасті, кругові, гістограми) підходять

для різних задач

- **Підграфіки** дозволяють поєднувати кілька візуалізацій на одному малюнку

- Графіки можна **налаштовувати** (кольори, стилі, підписи, легенди)
- Результати можна **зберігати** в різних форматах

У наступній лекції ми розглянемо створення графічних інтерфейсів користувача (GUI) за допомогою Tkinter та CustomTkinter.

Питання для обговорення

1. Поясніть базові команди для керування черепашкою в Turtle: forward(), backward(), right(), left(). Як налаштувати колір лінії та товщину пера?

2. Напишіть програму з використанням Turtle, яка малює квадрат. Потім модифікуйте її, щоб вона малювала правильний п'ятикутник. Як обчислити кут повороту для многокутника з n сторонами?

3. Поясніть різницю між командами penup() та pendown(). Для чого вони потрібні? Наведіть приклад, де потрібно перемістити черепашку без малювання лінії.

4. Проаналізуйте код:

```
import turtle
t = turtle.Turtle()
t.speed(0)
for i in range(36):
    for j in range(4):
        t.forward(100)
        t.right(90)
    t.right(10)
```

Що малює цей код? Як він працює? Що зміниться, якщо змінити зовнішній цикл на 72 ітерації, а внутрішній кут повороту на 5 градусів?

5. Що таке фрактали? Наведіть приклад рекурсивного малювання дерева або сніжинки Коха. Чому рекурсія добре підходить для створення фракталів?

6. Як встановити Matplotlib? Напишіть програму, яка створює простий лінійний графік функції $y = x^2$ для x від -10 до 10. Додайте заголовок, підписи осей та сітку.

7. Поясніть різницю між точковою діаграмою (scatter plot) та лінійним графіком. У яких випадках доцільніше використовувати кожен з них? Наведіть приклади.

8. Проаналізуйте код:

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y1 = [1, 4, 9, 16, 25]
y2 = [1, 2, 3, 4, 5]
plt.plot(x, y1, label='y = x2')
plt.plot(x, y2, label='y = x')
plt.legend()
plt.show()
```

Що робить функція `legend()`? Що станеться, якщо її не викликати?

9. Як створити стовпчасту діаграму (bar chart) для відображення продажів по місяцях? Як додати значення над стовпчиками? Як зробити горизонтальну стовпчасту діаграму?

10. Як створити кілька підграфіків на одному малюнку за допомогою `subplot()` або `subplots()`? Наведіть приклад, де на одному малюнку розміщено лінійний графік та стовпчасту діаграму поруч.

ЛЕКЦІЯ 18

Створення GUI з Tkinter та CustomTkinter

План лекції

1. Вступ до GUI та Tkinter: створення першого вікна.

Пояснимо, що таке GUI, чому це важливо. Pozнайомимос'я з Tkinter як стандартною бібліотекою Python. Навчимос'я створювати головне вікно, налаштовувати його розміри, заголовок, кольори. Розберемо цикл подій `mainloop()`.

2. Основні віджети: Label, Button, Entry.

Розглянемо базові елементи інтерфейсу: текстові мітки, кнопки, поля введення. Навчимос'я обробляти події (наприклад, натискання кнопок). Покажемо, як отримувати дані від користувача та виводити результати. Створимо просту програму-калькулятор.

3. Менеджери компоновання: pack, grid, place.

Пояснимо, як розміщувати віджети у вікні. Розберемо три менеджери компоновання: `pack` (простий послідовний), `grid` (табличний) та `place` (абсолютне позиціонування). Покажемо, коли який використовувати.

4. Розширені віджети та CustomTkinter.

Pozнайомимос'я з розширеними віджетами: `Listbox`, `Combobox`, `Scale`, `Checkbutton`, `Radiobutton`. А потім покажемо сучасну альтернативу – `CustomTkinter`, яка надає стильні, сучасні віджети з підтримкою тем оформлення. Створимо невеликий додаток із сучасним дизайном.

1. Вступ до GUI та Tkinter: створення першого вікна

Досі всі наші програми працювали в консолі: ми вводили текст, отримували текст. Але більшість сучасних програм мають графічний інтерфейс – вікна, кнопки, поля введення, меню. Створення таких програм називається **GUI програмуванням** (Graphical User Interface).

Чому GUI важливий?

- **Зручність** – користувачам не потрібно пам'ятати команди
- **Інтуїтивність** – елементи інтерфейсу підказують, що робити
- **Візуальна зворотна зв'язка** – результат видно одразу
- **Професійний вигляд** – програми виглядають як справжні додатки

Tkinter – стандартна бібліотека для GUI

Tkinter (скорочення від Tk interface) – це стандартна бібліотека Python для створення графічних інтерфейсів. Вона входить до складу Python, тому не

потребує додаткового встановлення. Tkinter проста у вивченні, але достатньо потужна для створення повноцінних додатків.

Перша програма: створення вікна

```
import tkinter as tk
```

```
# Створюємо головне вікно
```

```
root = tk.Tk()
```

```
# Запускаємо цикл обробки подій
```

```
root.mainloop()
```

Це мінімальна програма, яка створює порожнє вікно. Вона буде працювати, поки ми не закриємо вікно.

Налаштування вікна

```
import tkinter as tk
```

```
# Створюємо вікно
```

```
root = tk.Tk()
```

```
# Налаштовуємо вікно
```

```
root.title("Моя перша програма") # заголовок вікна
```

```
root.geometry("400x300") # розмір вікна (ширина x висота)
```

```
root.resizable(True, True) # чи можна змінювати розмір (ширина, висота)
```

```
root.minsize(300, 200) # мінімальний розмір
```

```
root.maxsize(600, 400) # максимальний розмір
```

```
root.configure(bg="lightblue") # колір фону
```

```
# Можна встановити іконку
```

```
# root.iconbitmap("icon.ico")
```

```
# Запускаємо
```

```
root.mainloop()
```

Додавання першого віджета – Label

```
import tkinter as tk
```

```

root = tk.Tk()
root.title("Приклад з міткою")
root.geometry("300x200")

# Створюємо текстову мітку
label = tk.Label(
    root, # батьківське вікно
    text="Привіт, світе!", # текст
    font=("Arial", 16, "bold"), # шрифт
    fg="blue", # колір тексту (foreground)
    bg="yellow", # колір фону (background)
    width=20, # ширина в символах
    height=2, # висота в рядках
    relief="solid", # тип рамки
    padx=10, # відступ по горизонталі
    pady=10 # відступ по вертикалі
)

```

```

# Розміщуємо мітку у вікні
label.pack(pady=20)

```

```

root.mainloop()

```

Додавання кнопки

```

import tkinter as tk

def button_click():
    """Функція, яка викликається при натисканні кнопки"""
    label.config(text="Кнопку натиснуто!", fg="red")

root = tk.Tk()
root.title("Приклад з кнопкою")
root.geometry("300x200")

# Створюємо мітку
label = tk.Label(root, text="Натисніть кнопку", font=("Arial", 12))

```

```
label.pack(pady=20)
```

```
# Створюємо кнопку
```

```
button = tk.Button(  
    root,  
    text="Натисни мене",  
    font=("Arial", 10),  
    bg="lightgreen",  
    fg="black",  
    padx=20,  
    pady=10,  
    command=button_click # функція, яка викликається при натисканні  
)  
button.pack()
```

```
root.mainloop()
```

Додавання поля введення

```
import tkinter as tk
```

```
def show_text():  
    """Показує введений текст"""  
    text = entry.get() # отримуємо текст з поля введення  
    result_label.config(text=f"Ви ввели: {text}")
```

```
def clear_text():  
    """Очищає поле введення"""  
    entry.delete(0, tk.END) # видаляємо текст з позиції 0 до кінця  
    result_label.config(text="")
```

```
root = tk.Tk()  
root.title("Приклад з полем введення")  
root.geometry("400x200")
```

```
# Створюємо мітку-підказку
```

```
prompt_label = tk.Label(root, text="Введіть текст:", font=("Arial", 10))  
prompt_label.pack(pady=5)
```

Створюємо поле введення

```
entry = tk.Entry(root, width=30, font=("Arial", 10))
entry.pack(pady=5)
```

Створюємо кнопки

```
show_button = tk.Button(root, text="Показати", command=show_text)
show_button.pack(pady=5)
```

```
clear_button = tk.Button(root, text="Очистити", command=clear_text)
clear_button.pack(pady=5)
```

Мітка для результату

```
result_label = tk.Label(root, text="", font=("Arial", 12, "bold"))
result_label.pack(pady=10)
```

```
root.mainloop()
```

Практичний приклад: простий калькулятор (перша версія)

```
import tkinter as tk
```

```
def calculate():
```

```
    """Обчислює результат"""
```

```
    try:
```

```
        num1 = float(entry1.get())
```

```
        num2 = float(entry2.get())
```

```
        operation = operation_var.get()
```

```
        if operation == "+":
```

```
            result = num1 + num2
```

```
        elif operation == "-":
```

```
            result = num1 - num2
```

```
        elif operation == "*":
```

```
            result = num1 * num2
```

```
        elif operation == "/":
```

```
            if num2 == 0:
```

```
                result_label.config(text="Помилка: ділення на нуль!")
```

```

        return
    result = num1 / num2
else:
    result_label.config(text="Оберіть операцію!")
    return

    result_label.config(text=f"Результат: {result:.2f}")
except ValueError:
    result_label.config(text="Помилка: введіть числа!")

def clear_fields():
    """Очищає всі поля"""
    entry1.delete(0, tk.END)
    entry2.delete(0, tk.END)
    result_label.config(text="")

root = tk.Tk()
root.title("Простий калькулятор")
root.geometry("400x350")
root.resizable(False, False)

# Заголовок
title_label = tk.Label(
    root,
    text="КАЛЬКУЛЯТОР",
    font=("Arial", 16, "bold"),
    fg="blue"
)
title_label.pack(pady=10)

# Перше число
frame1 = tk.Frame(root)
frame1.pack(pady=5)
tk.Label(frame1, text="Число 1:", font=("Arial", 10)).pack(side=tk.LEFT,
padx=5)
entry1 = tk.Entry(frame1, width=15, font=("Arial", 10))
entry1.pack(side=tk.LEFT)

```

```

# Друге число
frame2 = tk.Frame(root)
frame2.pack(pady=5)
tk.Label(frame2, text="Число 2:", font=("Arial", 10)).pack(side=tk.LEFT,
padx=5)
entry2 = tk.Entry(frame2, width=15, font=("Arial", 10))
entry2.pack(side=tk.LEFT)

# Вибір операції
frame3 = tk.Frame(root)
frame3.pack(pady=10)

operation_var = tk.StringVar(value="+") # змінна для зберігання вибраної
операції

operations = [("+", "+"), ("-", "-"), ("*", "*"), ("/", "/")]
for text, value in operations:
    tk.Radiobutton(
        frame3,
        text=text,
        variable=operation_var,
        value=value,
        font=("Arial", 10)
    ).pack(side=tk.LEFT, padx=10)

# Кнопки
button_frame = tk.Frame(root)
button_frame.pack(pady=10)

calc_button = tk.Button(
    button_frame,
    text="Обчислити",
    command=calculate,
    bg="lightgreen",
    padx=20,
    pady=5

```



```

)
calc_button.pack(side=tk.LEFT, padx=5)

clear_button = tk.Button(
    button_frame,
    text="Очистити",
    command=clear_fields,
    bg="lightcoral",
    padx=20,
    pady=5
)
clear_button.pack(side=tk.LEFT, padx=5)

# Результат
result_label = tk.Label(
    root,
    text="",
    font=("Arial", 14, "bold"),
    fg="darkgreen"
)
result_label.pack(pady=20)

root.mainloop()

```

2. Основні віджети: Label, Button, Entry

Розглянемо детальніше базові віджети Tkinter, їх властивості та методи.

Label – текстова мітка

```

import tkinter as tk

def label_demo():
    """Демонстрація всіх можливостей Label"""

    root = tk.Tk()
    root.title("Демонстрація Label")

```

```
root.geometry("500x400")
```

```
# Звичайний текст
```

```
tk.Label(root, text="1. Звичайний текст").pack(pady=5)
```

```
# Зміна шрифту
```

```
tk.Label(  
    root,  
    text="2. Різні шрифти",  
    font=("Times New Roman", 14, "italic")  
).pack(pady=5)
```

```
# Кольори
```

```
tk.Label(  
    root,  
    text="3. Кольоровий текст",  
    fg="red",  
    bg="yellow",  
    font=("Arial", 12, "bold")  
).pack(pady=5)
```

```
# Рамка
```

```
tk.Label(  
    root,  
    text="4. З рамкою",  
    relief="solid",  
    borderwidth=2,  
    padx=10,  
    pady=5  
).pack(pady=5)
```

```
# Різні типи рамок
```

```
frame = tk.Frame(root)  
frame.pack(pady=10)
```

```
reliefs = ["flat", "raised", "sunken", "ridge", "groove", "solid"]  
for i, relief in enumerate(reliefs):
```

```

tk.Label(
    frame,
    text=relief,
    relief=relief,
    borderwidth=2,
    width=10,
    height=2
).grid(row=0, column=i, padx=5)

```

Багаторядковий текст

```

tk.Label(
    root,
    text="5. Багаторядковий\ntекст з переносом\nна нові рядки",
    justify=tk.CENTER,
    bg="lightgray",
    padx=20,
    pady=10
).pack(pady=10)

```

```

root.mainloop()

```

label_demo()

Button – кнопка

```

import tkinter as tk

```

```

def button_demo():

```

```

    """Демонстрація всіх можливостей Button"""

```

```

    root = tk.Tk()
    root.title("Демонстрація Button")
    root.geometry("500x400")

```

```

    counter = 0

```

```

    def increment():
        nonlocal counter

```

```

counter += 1
counter_label.config(text=f"Натиснуто {counter} разів")

def change_state():
    if button2["state"] == "normal":
        button2.config(state="disabled", text="Вимкнено")
    else:
        button2.config(state="normal", text="Активна кнопка")

# Звичайна кнопка
tk.Label(root, text="1. Звичайна кнопка").pack()
button1 = tk.Button(root, text="Натисни мене", command=increment)
button1.pack(pady=5)

# Лічильник натискань
counter_label = tk.Label(root, text="Натиснуто 0 разів")
counter_label.pack(pady=5)

# Кнопка з різними стилями
tk.Label(root, text="2. Різні стилі кнопок").pack(pady=(20, 5))

frame = tk.Frame(root)
frame.pack()

# Колір фону
tk.Button(
    frame,
    text="Зелена",
    bg="lightgreen",
    fg="darkgreen",
    padx=20,
    pady=5
).grid(row=0, column=0, padx=5)

# Колір тексту
tk.Button(
    frame,

```

```
text="Червона",
fg="red",
padx=20,
pady=5
).grid(row=0, column=1, padx=5)
```

3 рамкою

```
tk.Button(
    frame,
    text="3 рамкою",
    relief="ridge",
    borderwidth=3,
    padx=20,
    pady=5
).grid(row=0, column=2, padx=5)
```

Вимкнена кнопка

```
tk.Button(
    frame,
    text="Вимкнена",
    state="disabled",
    padx=20,
    pady=5
).grid(row=0, column=3, padx=5)
```

Кнопка зі зміною стану

```
tk.Label(root, text="3. Кнопка зі зміною стану").pack(pady=(20, 5))
```

```
button2 = tk.Button(
    root,
    text="Активна кнопка",
    command=lambda: None,
    bg="lightblue"
)
button2.pack()
```

```
toggle_button = tk.Button(
```

```

    root,
    text="Увімкнути/Вимкнути",
    command=change_state
)
toggle_button.pack(pady=5)

root.mainloop()

```

button_demo()

Entry – поле введення

```
import tkinter as tk
```

```
def entry_demo():
```

```
    """Демонстрація всіх можливостей Entry"""
```

```

    root = tk.Tk()
    root.title("Демонстрація Entry")
    root.geometry("500x400")

```

```
def show_text():
```

```

    text = entry1.get()
    label1.config(text=f"Ви ввели: {text}")

```

```
def insert_text():
```

```
    entry1.insert(0, "Текст для вставки")
```

```
def clear_text():
```

```

    entry1.delete(0, tk.END)
    label1.config(text="")

```

```
def check_password():
```

```

    password = entry2.get()
    if password == "secret":
        label2.config(text="✓ Пароль правильний", fg="green")
    else:

```

```

label2.config(text="✗ Пароль неправильний", fg="red")

# Звичайне поле введення
tk.Label(root, text="1. Звичайне поле введення").pack(pady=5)
entry1 = tk.Entry(root, width=30)
entry1.pack()

# Кнопки для роботи з полем
frame1 = tk.Frame(root)
frame1.pack(pady=5)

tk.Button(frame1, text="Показати",
command=show_text).pack(side=tk.LEFT, padx=5)
tk.Button(frame1, text="Вставити",
command=insert_text).pack(side=tk.LEFT, padx=5)
tk.Button(frame1, text="Очистити",
command=clear_text).pack(side=tk.LEFT, padx=5)

label1 = tk.Label(root, text="")
label1.pack()

# Поле для пароля
tk.Label(root, text="\n2. Поле для пароля").pack()
entry2 = tk.Entry(root, show="*", width=30)
entry2.pack()

tk.Button(root, text="Перевірити пароль",
command=check_password).pack(pady=5)
label2 = tk.Label(root, text="")
label2.pack()

# Поле з підказкою
tk.Label(root, text="\n3. Поле з підказкою (placeholder)").pack()

def on_entry_click(event):
    if entry3.get() == "Введіть текст...":
        entry3.delete(0, tk.END)

```

```

entry3.config(fg="black")

def on_focus_out(event):
    if entry3.get() == "":
        entry3.insert(0, "Введіть текст...")
        entry3.config(fg="gray")

entry3 = tk.Entry(root, width=30, fg="gray")
entry3.insert(0, "Введіть текст...")
entry3.bind("<FocusIn>", on_entry_click)
entry3.bind("<FocusOut>", on_focus_out)
entry3.pack(pady=5)

# Поле тільки для читання
tk.Label(root, text="\n4. Поле тільки для читання").pack()
entry4 = tk.Entry(root, width=30, state="readonly")
entry4.insert(0, "Цей текст не можна змінити")
entry4.pack(pady=5)

root.mainloop()

# entry_demo()

```

Frame – контейнер для організації віджетів

```

import tkinter as tk

def frame_demo():
    """Демонстрація використання Frame"""

    root = tk.Tk()
    root.title("Демонстрація Frame")
    root.geometry("500x400")

    # Створюємо рамку з рамкою
    frame1 = tk.Frame(
        root,
        relief="ridge",

```



```

        borderwidth=3,
        bg="lightblue"
    )
    frame1.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

# Заголовок всередині рамки
    tk.Label(
        frame1,
        text="Контактна інформація",
        font=("Arial", 12, "bold"),
        bg="lightblue"
    ).pack(pady=5)

# Створюємо внутрішню рамку для полів введення
    frame2 = tk.Frame(frame1, bg="lightblue")
    frame2.pack(pady=10)

# Поля введення
    tk.Label(frame2, text="Ім'я:", bg="lightblue").grid(row=0, column=0,
    padx=5, pady=5)
    entry1 = tk.Entry(frame2, width=20)
    entry1.grid(row=0, column=1, padx=5, pady=5)

    tk.Label(frame2, text="Прізвище:", bg="lightblue").grid(row=1, column=0,
    padx=5, pady=5)
    entry2 = tk.Entry(frame2, width=20)
    entry2.grid(row=1, column=1, padx=5, pady=5)

    tk.Label(frame2, text="Email:", bg="lightblue").grid(row=2, column=0,
    padx=5, pady=5)
    entry3 = tk.Entry(frame2, width=20)
    entry3.grid(row=2, column=1, padx=5, pady=5)

# Рамка для кнопок
    frame3 = tk.Frame(frame1, bg="lightblue")
    frame3.pack(pady=10)

```

```
tk.Button(
    frame3,
    text="Зберегти",
    bg="lightgreen",
    padx=20
).pack(side=tk.LEFT, padx=5)
```

```
tk.Button(
    frame3,
    text="Скасувати",
    bg="lightcoral",
    padx=20
).pack(side=tk.LEFT, padx=5)
```

```
root.mainloop()
```

```
# frame_demo()
```

Message – багаторядкове повідомлення

```
import tkinter as tk
```

```
def message_demo():
```

```
    """Демонстрація Message віджета"""
```

```
    root = tk.Tk()
```

```
    root.title("Демонстрація Message")
```

```
    root.geometry("500x400")
```

```
    long_text = """Це дуже довгий текст, який не поміститься в один рядок.
    Віджет Message автоматично переносить текст на нові рядки,
    щоб він помістився в задану ширину. Це зручно для виведення
    довгих повідомлень або інструкцій."""
```

```
# Message з автоматичним переносом
```

```
msg1 = tk.Message(
    root,
    text=long_text,
```

```

        width=300,
        font=("Arial", 10),
        justify=tk.LEFT,
        relief="solid",
        borderwidth=2,
        padx=10,
        pady=10
    )
    msg1.pack(pady=10)

# Message з вирівнюванням по центру
    msg2 = tk.Message(
        root,
        text=long_text,
        width=300,
        font=("Arial", 10),
        justify=tk.CENTER,
        bg="lightyellow",
        padx=10,
        pady=10
    )
    msg2.pack(pady=10)

    root.mainloop()

```

message_demo()

Практичний приклад: програма для реєстрації користувача

```

import tkinter as tk
from tkinter import messagebox

class RegistrationForm:
    """Клас для форми реєстрації"""

    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Реєстрація користувача")

```

```
self.root.geometry("400x500")
self.root.resizable(False, False)
```

```
self.create_widgets()
```

```
def create_widgets(self):
```

```
    # Заголовок
```

```
    title_label = tk.Label(
        self.root,
        text="👤 ФОРМА РЕЄСТРАЦІЇ",
        font=("Arial", 16, "bold"),
        fg="darkblue"
```

```
    )
    title_label.pack(pady=20)
```

```
    # Рамка для полів введення
```

```
    frame = tk.Frame(self.root, bg="#f0f0f0", relief="ridge", borderwidth=2)
    frame.pack(padx=20, pady=10, fill=tk.BOTH, expand=True)
```

```
    # Ім'я
```

```
    tk.Label(frame, text="Ім'я:", bg="#f0f0f0", font=("Arial", 10)).grid(
        row=0, column=0, padx=10, pady=10, sticky="e"
    )
```

```
    self.name_entry = tk.Entry(frame, width=25, font=("Arial", 10))
    self.name_entry.grid(row=0, column=1, padx=10, pady=10)
```

```
    # Прізвище
```

```
    tk.Label(frame, text="Прізвище:", bg="#f0f0f0", font=("Arial", 10)).grid(
        row=1, column=0, padx=10, pady=10, sticky="e"
    )
```

```
    self.surname_entry = tk.Entry(frame, width=25, font=("Arial", 10))
    self.surname_entry.grid(row=1, column=1, padx=10, pady=10)
```

```
    # Email
```

```
    tk.Label(frame, text="Email:", bg="#f0f0f0", font=("Arial", 10)).grid(
        row=2, column=0, padx=10, pady=10, sticky="e"
    )
```

```

self.email_entry = tk.Entry(frame, width=25, font=("Arial", 10))
self.email_entry.grid(row=2, column=1, padx=10, pady=10)

# Вік
tk.Label(frame, text="Вік:", bg="#f0f0f0", font=("Arial", 10)).grid(
    row=3, column=0, padx=10, pady=10, sticky="e"
)
self.age_entry = tk.Entry(frame, width=25, font=("Arial", 10))
self.age_entry.grid(row=3, column=1, padx=10, pady=10)

# Стать
tk.Label(frame, text="Стать:", bg="#f0f0f0", font=("Arial", 10)).grid(
    row=4, column=0, padx=10, pady=10, sticky="e"
)

self.gender_var = tk.StringVar(value="Не вказано")
gender_frame = tk.Frame(frame, bg="#f0f0f0")
gender_frame.grid(row=4, column=1, padx=10, pady=10, sticky="w")

tk.Radiobutton(
    gender_frame,
    text="Чоловіча",
    variable=self.gender_var,
    value="Чоловіча",
    bg="#f0f0f0"
).pack(side=tk.LEFT, padx=5)

tk.Radiobutton(
    gender_frame,
    text="Жіноча",
    variable=self.gender_var,
    value="Жіноча",
    bg="#f0f0f0"
).pack(side=tk.LEFT, padx=5)

# Країна
tk.Label(frame, text="Країна:", bg="#f0f0f0", font=("Arial", 10)).grid(

```

```

        row=5, column=0, padx=10, pady=10, sticky="e"
    )

    countries = ["Україна", "Польща", "Німеччина", "США", "Інша"]
    self.country_var = tk.StringVar(value="Україна")
    country_menu = tk.OptionMenu(frame, self.country_var, *countries)
    country_menu.config(width=20)
    country_menu.grid(row=5, column=1, padx=10, pady=10)

    # Отримувати новини
    self.news_var = tk.BooleanVar(value=True)
    tk.Checkbutton(
        frame,
        text="Отримувати новини на email",
        variable=self.news_var,
        bg="#f0f0f0"
    ).grid(row=6, column=0, columnspan=2, pady=10)

    # Кнопки
    button_frame = tk.Frame(self.root)
    button_frame.pack(pady=20)

    tk.Button(
        button_frame,
        text="Зареєструватися",
        command=self.register,
        bg="lightgreen",
        padx=20,
        pady=5,
        font=("Arial", 10, "bold")
    ).pack(side=tk.LEFT, padx=10)

    tk.Button(
        button_frame,
        text="Очистити",
        command=self.clear_form,
        bg="lightcoral",

```

```

    padx=20,
    pady=5,
    font=("Arial", 10, "bold")
).pack(side=tk.LEFT, padx=10)

```

Мітка для результатів

```

self.result_label = tk.Label(
    self.root,
    text="",
    font=("Arial", 10),
    fg="darkgreen",
    wraplength=350
)
self.result_label.pack(pady=10)

```

def register(self):

"""Обробка реєстрації"""

Отримуємо дані

```

name = self.name_entry.get().strip()
surname = self.surname_entry.get().strip()
email = self.email_entry.get().strip()
age = self.age_entry.get().strip()
gender = self.gender_var.get()
country = self.country_var.get()
get_news = "Так" if self.news_var.get() else "Ні"

```

Валідація

if not name or not surname or not email:

messagebox.showerror("Помилка", "Будь ласка, заповніть обов'язкові поля")

return

if not self.validate_email(email):

messagebox.showerror("Помилка", "Введіть коректний email")

return

Формуємо результат

```

result = f"✓ Реєстрація успішна!\n\n"
result += f"Ім'я: {name} {surname}\n"
result += f"Email: {email}\n"
if age:
    result += f"Вік: {age}\n"
result += f"Стать: {gender}\n"
result += f"Країна: {country}\n"
result += f"Отримувати новини: {get_news}"

self.result_label.config(text=result)
messagebox.showinfo("Успіх", "Реєстрацію завершено!")

```

```

def validate_email(self, email):
    """Проста валідація email"""
    return "@" in email and "." in email

```

```

def clear_form(self):
    """Очищає форму"""
    self.name_entry.delete(0, tk.END)
    self.surname_entry.delete(0, tk.END)
    self.email_entry.delete(0, tk.END)
    self.age_entry.delete(0, tk.END)
    self.gender_var.set("Не вказано")
    self.country_var.set("Україна")
    self.news_var.set(True)
    self.result_label.config(text="")

```

```

def run(self):
    """Запускає додаток"""
    self.root.mainloop()

```

Запуск програми

```

if __name__ == "__main__":
    app = RegistrationForm()
    app.run()

```

Підсумки

- **Tkinter** – стандартна бібліотека для створення GUI в Python

- **Головне вікно** створюється через `tk.Tk()`, цикл подій – через `mainloop()`
- **Label** використовується для відображення тексту
- **Button** – для створення кнопок з обробниками подій
- **Entry** – для введення тексту користувачем
- **Frame** – для групування віджетів
- **Змінні Tkinter** (`StringVar`, `IntVar`, `BooleanVar`) зберігають стан віджетів

У наступному пункті ми розглянемо менеджери компоновання (`pack`, `grid`, `place`), які дозволяють розміщувати віджети у вікні більш гнучко.

3. Менеджери компоновання: `pack`, `grid`, `place`

Менеджери компоновання (`geometry managers`) визначають, як віджети розташовуються у вікні або контейнері. Tkinter надає три менеджери: `pack`, `grid` та `place`. Кожен має свої переваги і підходить для різних ситуацій.

1. `Pack` – простий послідовний менеджер

`pack` розміщує віджети один за одним у порядку додавання. Це найпростіший менеджер, ідеальний для лінійних макетів.

```
import tkinter as tk

def pack_demo():
    """Демонстрація різних режимів pack"""

    root = tk.Tk()
    root.title("Демонстрація pack")
    root.geometry("400x300")

    # Створюємо рамки для різних прикладів
    frame1 = tk.Frame(root, relief="ridge", borderwidth=2)
    frame1.pack(padx=10, pady=10, fill=tk.X)

    tk.Label(frame1, text="1. Звичайний pack (зверху вниз):").pack(anchor=tk.W)

    for i in range(3):
        tk.Button(frame1, text=f"Кнопка {i+1}", width=15).pack(pady=2)
```

```

# Pack зліва направо
frame2 = tk.Frame(root, relief="ridge", borderwidth=2)
frame2.pack(padx=10, pady=10, fill=tk.X)

tk.Label(frame2, text="2. Pack зліва направо:").pack(anchor=tk.W)

for i in range(3):
    tk.Button(frame2, text=f"Кнопка {i+1}", width=10).pack(side=tk.LEFT,
padx=5, pady=5)

# Pack з різними параметрами expand/fill
frame3 = tk.Frame(root, relief="ridge", borderwidth=2, height=100)
frame3.pack(padx=10, pady=10, fill=tk.X)
frame3.pack_propagate(False) # забороняємо змінювати розмір рамки

tk.Label(frame3, text="3. 3 параметрами expand та
fill:").pack(anchor=tk.W)

tk.Button(frame3, text="expand=True, fill=BOTH", bg="lightblue").pack(
    expand=True, fill=tk.BOTH, padx=5, pady=5
)

root.mainloop()

# pack_demo()

```

Параметри pack:

Параметр	Опис
side	Сторона для розміщення: tk.TOP (за замовчуванням), tk.BOTTOM, tk.LEFT, tk.RIGHT
fill	Чи заповнювати доступний простір: tk.NONE, tk.X, tk.Y, tk.BOTH
expand	Чи розширюватися при збільшенні вікна (True/False)
padx/pady	Зовнішні відступи по горизонталі/вертикалі

Параметр	Опис
ipadx/ipady	Внутрішні відступи

2. Grid – табличний менеджер

grid розміщує віджети у вигляді таблиці з рядками та стовпцями. Це найпотужніший і найзручніший менеджер для складних макетів.

```
import tkinter as tk

def grid_demo():
    """Демонстрація різних можливостей grid"""

    root = tk.Tk()
    root.title("Демонстрація grid")
    root.geometry("500x400")

    # Проста сітка
    frame1 = tk.Frame(root, relief="ridge", borderwidth=2)
    frame1.pack(padx=10, pady=10, fill=tk.X)

    tk.Label(frame1, text="1. Проста сітка 3x3:").grid(row=0, column=0,
columnspan=3, pady=5)

    for i in range(3):
        for j in range(3):
            tk.Button(frame1, text=f"({i},{j})", width=8, height=2).grid(
                row=i+1, column=j, padx=2, pady=2
            )

    # Форма введення з підписами
    frame2 = tk.Frame(root, relief="ridge", borderwidth=2)
    frame2.pack(padx=10, pady=10, fill=tk.X)

    tk.Label(frame2, text="2. Форма введення:").grid(row=0, column=0,
columnspan=2, pady=5)
```

```
labels = ["Ім'я:", "Прізвище:", "Email:", "Телефон:"]
entries = []
```

```
for i, label_text in enumerate(labels):
    tk.Label(frame2, text=label_text, width=10, anchor=tk.E).grid(
        row=i+1, column=0, padx=5, pady=5, sticky=tk.E
    )
    entry = tk.Entry(frame2, width=30)
    entry.grid(row=i+1, column=1, padx=5, pady=5, sticky=tk.W)
    entries.append(entry)
```

```
# Кнопки внизу форми
```

```
tk.Button(frame2, text="Зберегти", bg="lightgreen").grid(
    row=len(labels)+1, column=0, padx=5, pady=10, sticky=tk.E
)
tk.Button(frame2, text="Скасувати", bg="lightcoral").grid(
    row=len(labels)+1, column=1, padx=5, pady=10, sticky=tk.W
)
```

```
# Складний макет з об'єднанням комірок
```

```
frame3 = tk.Frame(root, relief="ridge", borderwidth=2)
frame3.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)
```

```
tk.Label(frame3, text="3. Складний макет з об'єднанням:").grid(
    row=0, column=0, columnspan=3, pady=5
)
```

```
# Заголовок
```

```
tk.Label(frame3, text="Заголовок", bg="lightblue", height=2).grid(
    row=1, column=0, columnspan=3, sticky=tk.EW, padx=2, pady=2
)
```

```
# Ліва панель
```

```
tk.Label(frame3, text="Ліва панель", bg="lightgreen", width=15,
height=5).grid(
    row=2, column=0, sticky=tk.NSEW, padx=2, pady=2
)
```

```

# Центральна панель
tk.Label(frame3, text="Центральна панель", bg="yellow", height=5).grid(
    row=2, column=1, sticky=tk.NSEW, padx=2, pady=2
)

# Права панель
tk.Label(frame3, text="Права панель", bg="lightcoral", width=15,
height=5).grid(
    row=2, column=2, sticky=tk.NSEW, padx=2, pady=2
)

# Футер
tk.Label(frame3, text="Футер", bg="lightgray", height=2).grid(
    row=3, column=0, columnspan=3, sticky=tk.EW, padx=2, pady=2
)

# Налаштовуємо ваги стовпців для розтягування
frame3.grid_columnconfigure(1, weight=1) # центральний стовець
розтягується
frame3.grid_rowconfigure(2, weight=1) # центральний рядок
розтягується

root.mainloop()

# grid_demo()

```

Параметри grid:

Параметр	Опис
row/column	Рядок і стовець для розміщення
rowspan/columnspan	Скільки рядків/стовпців займає віджет
sticky	До яких сторін "прилипає": tk.N, tk.S, tk.E, tk.W, комбінації (tk.NSEW)
padx/pady	Зовнішні відступи

Параметр	Опис
ipadx/ipady	Внутрішні відступи

3. Place – абсолютне позиціонування

place дозволяє розміщувати віджети за точними координатами. Це найменш гнучкий менеджер, але корисний для специфічних випадків.

```
import tkinter as tk

def place_demo():
    """Демонстрація place"""

    root = tk.Tk()
    root.title("Демонстрація place")
    root.geometry("400x300")

    # Абсолютне позиціонування
    frame = tk.Frame(root, relief="ridge", borderwidth=2)
    frame.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

    tk.Label(frame, text="Абсолютне позиціонування (x, y):").place(x=10,
y=10)

    tk.Button(frame, text="Кнопка 1", bg="lightblue").place(x=50, y=50)
    tk.Button(frame, text="Кнопка 2", bg="lightgreen").place(x=150, y=100)
    tk.Button(frame, text="Кнопка 3", bg="lightcoral").place(x=250, y=150)

    # Відносне позиціонування
    frame2 = tk.Frame(root, relief="ridge", borderwidth=2)
    frame2.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

    tk.Label(frame2, text="Відносне позиціонування (relx,
rely):").place(relx=0.1, rely=0.1)

    tk.Button(frame2, text="Центр", bg="yellow").place(relx=0.5, rely=0.5,
anchor=tk.CENTER)
```

```
tk.Button(frame2, text="Верхній лівий", bg="lightblue").place(relx=0.1,
rely=0.3)
tk.Button(frame2, text="Нижній правий", bg="lightgreen").place(relx=0.7,
rely=0.7)
```

```
root.mainloop()
```

```
# place_demo()
```

Параметри place:

Параметр	Опис
x/y	Абсолютні координати в пікселях
relx/rely	Відносні координати (0.0 - 1.0)
width/height	Ширина і висота в пікселях
relwidth/relheight	Відносна ширина і висота (0.0 - 1.0)
anchor	Точка прив'язки: tk.NW, tk.CENTER, tk.SE і т.д.

Порівняння менеджерів компоновання

```
import tkinter as tk
```

```
def layout_comparison():
```

```
    """Порівняння різних менеджерів компоновання"""
```

```
    root = tk.Tk()
```

```
    root.title("Порівняння менеджерів")
```

```
    root.geometry("600x400")
```

```
# Створюємо три рамки для демонстрації
```

```
    frame1 = tk.Frame(root, relief="ridge", borderwidth=2)
```

```
    frame1.pack(side=tk.LEFT, padx=10, pady=10, fill=tk.BOTH, expand=True)
```

```
    frame2 = tk.Frame(root, relief="ridge", borderwidth=2)
```

```
frame2.pack(side=tk.LEFT, padx=10, pady=10, fill=tk.BOTH, expand=True)
```

```
frame3 = tk.Frame(root, relief="ridge", borderwidth=2)
```

```
frame3.pack(side=tk.LEFT, padx=10, pady=10, fill=tk.BOTH, expand=True)
```

Заголовки

```
tk.Label(frame1, text="PACK", font=("Arial", 12, "bold")).pack(pady=5)
```

```
tk.Label(frame2, text="GRID", font=("Arial", 12, "bold")).pack(pady=5)
```

```
tk.Label(frame3, text="PLACE", font=("Arial", 12, "bold")).pack(pady=5)
```

Приклад з pack

```
tk.Label(frame1, text="Зверху").pack()
```

```
tk.Button(frame1, text="Кнопка 1").pack(fill=tk.X, padx=10, pady=2)
```

```
tk.Button(frame1, text="Кнопка 2").pack(fill=tk.X, padx=10, pady=2)
```

```
tk.Button(frame1, text="Кнопка 3").pack(fill=tk.X, padx=10, pady=2)
```

```
tk.Label(frame1, text="Знизу").pack(side=tk.BOTTOM)
```

Приклад з grid

```
for i in range(3):
```

```
    for j in range(2):
```

```
        tk.Button(frame2, text=f"({i},{j})", width=8).grid(
            row=i, column=j, padx=2, pady=2, sticky=tk.NSEW
        )
```

Налаштовуємо ваги для grid

```
for i in range(3):
```

```
    frame2.grid_rowconfigure(i, weight=1)
```

```
for j in range(2):
```

```
    frame2.grid_columnconfigure(j, weight=1)
```

Приклад з place

```
tk.Button(frame3, text="Центр", bg="lightblue").place(relx=0.5, rely=0.5,
anchor=tk.CENTER)
```

```
tk.Button(frame3, text="NW", bg="lightgreen").place(relx=0.1, rely=0.1)
```

```
tk.Button(frame3, text="NE", bg="lightgreen").place(relx=0.9, rely=0.1,
anchor=tk.NE)
```



```

tk.Button(frame3, text="SW", bg="lightcoral").place(relx=0.1, rely=0.9,
anchor=tk.SW)
tk.Button(frame3, text="SE", bg="lightcoral").place(relx=0.9, rely=0.9,
anchor=tk.SE)

root.mainloop()

# layout_comparison()

```

Коли який менеджер використовувати?

Менеджер	Коли використовувати
pack	Прості лінійні макети, панелі інструментів, послідовність віджетів
grid	Складні табличні макети, форми введення, панелі з сіткою
place	Точне позиціонування, графічні редактори, специфічні макети

4. Розширені віджети та CustomTkinter

Окрім базових віджетів, Tkinter має багато розширених елементів. А CustomTkinter пропонує сучасні, стильні альтернативи з підтримкою тем.

Розширені віджети Tkinter

```

import tkinter as tk
from tkinter import ttk, messagebox, filedialog, colorchooser

def advanced_widgets():
    """Демонстрація розширених віджетів Tkinter"""

    root = tk.Tk()
    root.title("Розширені віджети")
    root.geometry("600x500")

    # Нотбук (вкладки)
    notebook = ttk.Notebook(root)

```

```

notebook.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

# Вкладка 1: Listbox та Combobox
tab1 = ttk.Frame(notebook)
notebook.add(tab1, text="Списки")

tk.Label(tab1, text="Listbox (список вибору):").pack(pady=5)

listbox = tk.Listbox(tab1, height=5)
listbox.pack(padx=10, pady=5, fill=tk.X)

for i in ["Python", "Java", "JavaScript", "C++", "Ruby"]:
    listbox.insert(tk.END, i)

tk.Label(tab1, text="Combobox (випадаючий список):").pack(pady=5)

combobox = ttk.Combobox(tab1, values=["Україна", "Польща",
"Німеччина", "США"])
combobox.pack(padx=10, pady=5)
combobox.set("Оберіть країну")

def show_selection():
    messagebox.showinfo("Вибір",
                        f"Listbox: {listbox.get(tk.ACTIVE)}\nCombobox: {combobox.get()}")

tk.Button(tab1, text="Показати вибір",
command=show_selection).pack(pady=10)

# Вкладка 2: Scale та Spinbox
tab2 = ttk.Frame(notebook)
notebook.add(tab2, text="Регулятори")

tk.Label(tab2, text="Scale (повзунок):").pack(pady=5)

scale_var = tk.IntVar(value=50)
scale = tk.Scale(tab2, from_=0, to=100, orient=tk.HORIZONTAL,

```

```

        variable=scale_var, length=200)
scale.pack(pady=5)

tk.Label(tab2, text="Spinbox (лічильник):").pack(pady=5)

spin_var = tk.IntVar(value=10)
spinbox = tk.Spinbox(tab2, from_=0, to=100, textvariable=spin_var,
width=10)
spinbox.pack(pady=5)

def show_values():
    messagebox.showinfo("Значення", f"Scale: {scale_var.get()}\nSpinbox:
{spin_var.get()}")

tk.Button(tab2, text="Показати значення",
command=show_values).pack(pady=10)

# Вкладка 3: Treeview (таблиця)
tab3 = ttk.Frame(notebook)
notebook.add(tab3, text="Таблиця")

tree = ttk.Treeview(tab3, columns=("name", "age", "grade"),
show="headings")
tree.heading("name", text="Ім'я")
tree.heading("age", text="Вік")
tree.heading("grade", text="Оцінка")

data = [
    ("Олена", 19, 92),
    ("Іван", 20, 85),
    ("Марія", 18, 95),
    ("Петро", 21, 78)
]

for item in data:
    tree.insert("", tk.END, values=item)

```

```

tree.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

# Вкладка 4: Діалогові вікна
tab4 = ttk.Frame(notebook)
notebook.add(tab4, text="Діалоги")

def open_file():
    filename = filedialog.askopenfilename(title="Виберіть файл")
    if filename:
        messagebox.showinfo("Вибрано файл", f"Ви вибрали:\n{filename}")

def save_file():
    filename = filedialog.asksaveasfilename(title="Зберегти файл")
    if filename:
        messagebox.showinfo("Збереження", f"Файл буде збережено\nяк:\n{filename}")

def choose_color():
    color = colorchooser.askcolor(title="Виберіть колір")
    if color[1]:
        messagebox.showinfo("Колір", f"Ви вибрали колір:\n{color[1]}")

tk.Button(tab4, text="Відкрити файл", command=open_file).pack(pady=5)
tk.Button(tab4, text="Зберегти файл", command=save_file).pack(pady=5)
tk.Button(tab4, text="Вибрати колір", command=choose_color).pack(pady=5)

root.mainloop()

# advanced_widgets()

```

Вступ до CustomTkinter

CustomTkinter – це сучасна бібліотека на основі Tkinter, яка надає стильні, сучасні віджети з підтримкою тем (світла/темна). Вона значно покращує зовнішній вигляд додатків.

Встановлення: pip install customtkinter

```

import customtkinter as ctk

def customtkinter_demo():
    """Демонстрація CustomTkinter"""

    # Налаштування зовнішнього вигляду
    ctk.set_appearance_mode("dark") # "dark", "light", "system"
    ctk.set_default_color_theme("blue") # "blue", "green", "dark-blue"

    # Створення вікна
    root = ctk.CTk()
    root.title("CustomTkinter Demo")
    root.geometry("500x600")

    # Заголовок
    title_label = ctk.CTkLabel(
        root,
        text="CustomTkinter - Сучасний GUI",
        font=ctk.CTkFont(size=20, weight="bold")
    )
    title_label.pack(pady=20)

    # Рамка для віджетів
    frame = ctk.CTkFrame(root)
    frame.pack(padx=20, pady=10, fill=tk.BOTH, expand=True)

    # Звичайна кнопка
    def button_callback():
        result_label.configure(text=f"Привіт, {entry.get()}!")

    ctk.CTkLabel(frame, text="Введіть ім'я:").pack(pady=5)
    entry = ctk.CTkEntry(frame, placeholder_text="Наприклад: Олена",
width=200)
    entry.pack(pady=5)

    button = ctk.CTkButton(
        frame,

```

```

text="Натисни мене",
command=button_callback,
width=200,
height=40,
corner_radius=10
)
button.pack(pady=10)

```

```

result_label = ctk.CTkLabel(frame, text="")
result_label.pack(pady=5)

```

Перемикач теми

```

def switch_theme():
    if switch.get() == 1:
        ctk.set_appearance_mode("dark")
    else:
        ctk.set_appearance_mode("light")

```

```

switch = ctk.CTkSwitch(frame, text="Темна тема",
command=switch_theme)
switch.pack(pady=10)
switch.select() # увімкнено за замовчуванням

```

Прогрес бар

```

progressbar = ctk.CTkProgressBar(frame, width=200)
progressbar.pack(pady=10)
progressbar.set(0.5)

```

Слайдер

```

slider = ctk.CTkSlider(frame, from_=0, to=100, width=200)
slider.pack(pady=10)
slider.set(50)

```

Радіокнопки

```

radio_var = ctk.StringVar(value="option1")

radio_frame = ctk.CTkFrame(frame)

```

```
radio_frame.pack(pady=10)
```

```
ctk.CTkRadioButton(  
    radio_frame,  
    text="Варіант 1",  
    variable=radio_var,  
    value="option1"  
).pack(side=tk.LEFT, padx=10)
```

```
ctk.CTkRadioButton(  
    radio_frame,  
    text="Варіант 2",  
    variable=radio_var,  
    value="option2"  
).pack(side=tk.LEFT, padx=10)
```

```
# Комбобокс
```

```
combobox = ctk.CTkComboBox(  
    frame,  
    values=["Україна", "Польща", "Німеччина", "США"],  
    width=200  
)  
combobox.pack(pady=10)  
combobox.set("Україна")
```

```
# Чекбокс
```

```
checkbox = ctk.CTkCheckBox(frame, text="Погоджуюся з умовами")  
checkbox.pack(pady=10)
```

```
# Текстове поле
```

```
textbox = ctk.CTkTextbox(frame, width=300, height=100)  
textbox.pack(pady=10)  
textbox.insert("0.0", "Це багаторядкове текстове поле...")
```

```
root.mainloop()
```

```
# customtkinter_demo()
```

Практичний приклад: Сучасний додаток для нотаток

```
import customtkinter as ctk
from tkinter import messagebox
import json
import os

class NotesApp:
    """Сучасний додаток для нотаток з використанням CustomTkinter"""

    def __init__(self):
        # Налаштування теми
        ctk.set_appearance_mode("dark")
        ctk.set_default_color_theme("blue")

        self.root = ctk.CTk()
        self.root.title("📝 Мої нотатки")
        self.root.geometry("800x600")

        self.notes_file = "notes.json"
        self.current_note = None
        self.load_notes()
        self.create_widgets()

    def load_notes(self):
        """Завантажує нотатки з файлу"""
        if os.path.exists(self.notes_file):
            try:
                with open(self.notes_file, 'r', encoding='utf-8') as f:
                    self.notes = json.load(f)
            except:
                self.notes = []
        else:
            self.notes = []

    def save_notes(self):
        """Зберігає нотатки у файл"""
```



```
with open(self.notes_file, 'w', encoding='utf-8') as f:
    json.dump(self.notes, f, ensure_ascii=False, indent=4)
```

```
def create_widgets(self):
```

```
    # Головний контейнер
```

```
    main_frame = ctk.CTkFrame(self.root)
```

```
    main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)
```

```
    # Ліва панель - список нотаток
```

```
    left_frame = ctk.CTkFrame(main_frame, width=250)
```

```
    left_frame.pack(side=tk.LEFT, fill=tk.Y, padx=(0, 10))
```

```
    left_frame.pack_propagate(False)
```

```
    ctk.CTkLabel(
```

```
        left_frame,
```

```
        text="📁 Список нотаток",
```

```
        font=ctk.CTkFont(size=16, weight="bold")
```

```
    ).pack(pady=10)
```

```
    # Список нотаток
```

```
    self.notes_listbox = tk.Listbox(
```

```
        left_frame,
```

```
        bg="#2b2b2b",
```

```
        fg="white",
```

```
        selectbackground="#1f538d",
```

```
        font=("Arial", 11)
```

```
    )
```

```
    self.notes_listbox.pack(fill=tk.BOTH, expand=True, padx=10, pady=5)
```

```
    self.notes_listbox.bind('<<ListboxSelect>>', self.on_note_select)
```

```
    # Кнопки для списку
```

```
    btn_frame = ctk.CTkFrame(left_frame)
```

```
    btn_frame.pack(fill=tk.X, padx=10, pady=10)
```

```
    ctk.CTkButton(
```

```
        btn_frame,
```

```
        text="+ Нова нотатка",
```

```

        command=self.new_note
    ).pack(fill=tk.X, pady=2)

    ctk.CTkButton(
        btn_frame,
        text="🗑 Видалити",
        command=self.delete_note,
        fg_color="#e74c3c",
        hover_color="#c0392b"
    ).pack(fill=tk.X, pady=2)

# Права панель - редагування нотатки
    right_frame = ctk.CTkFrame(main_frame)
    right_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

# Заголовок
    title_frame = ctk.CTkFrame(right_frame)
    title_frame.pack(fill=tk.X, padx=10, pady=10)

    ctk.CTkLabel(title_frame, text="Заголовок:",
font=ctk.CTkFont(size=12)).pack(
        side=tk.LEFT, padx=(0, 10)
    )

    self.title_entry = ctk.CTkEntry(title_frame, placeholder_text="Введіть
заголовок")
    self.title_entry.pack(side=tk.LEFT, fill=tk.X, expand=True)

# Текст нотатки
    ctk.CTkLabel(right_frame, text="Текст нотатки:",
font=ctk.CTkFont(size=12)).pack(
        anchor=tk.W, padx=10, pady=(10, 0)
    )

    self.content_text = ctk.CTkTextbox(right_frame, height=300)
    self.content_text.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

```

Кнопки збереження

```
save_frame = ctk.CTkFrame(right_frame)
save_frame.pack(fill=tk.X, padx=10, pady=10)
```

```
ctk.CTkButton(
    save_frame,
    text="💾 Зберегти",
    command=self.save_note,
    fg_color="#27ae60",
    hover_color="#229954"
).pack(side=tk.LEFT, padx=5)
```

```
ctk.CTkButton(
    save_frame,
    text="🔄 Очистити",
    command=self.clear_form,
    fg_color="#f39c12",
    hover_color="#e67e22"
).pack(side=tk.LEFT, padx=5)
```

Інформація про кількість нотаток

```
self.info_label = ctk.CTkLabel(
    right_frame,
    text=f"Всього нотаток: {len(self.notes)}",
    font=ctk.CTkFont(size=10)
)
self.info_label.pack(pady=5)
```

Оновлюємо список

```
self.update_notes_list()
```

```
def update_notes_list(self):
    """Оновлює список нотаток"""
    self.notes_listbox.delete(0, tk.END)
    for note in self.notes:
        self.notes_listbox.insert(tk.END, note.get('title', 'Без назви'))
    self.info_label.configure(text=f"Всього нотаток: {len(self.notes)}")
```

```

def new_note(self):
    """Створює нову нотатку"""
    self.clear_form()
    self.current_note = None

def on_note_select(self, event):
    """Обробка вибору нотатки зі списку"""
    selection = self.notes_listbox.curselection()
    if selection:
        index = selection[0]
        self.current_note = index
        note = self.notes[index]
        self.title_entry.delete(0, tk.END)
        self.title_entry.insert(0, note.get('title', ''))
        self.content_text.delete('0.0', tk.END)
        self.content_text.insert('0.0', note.get('content', ''))

def save_note(self):
    """Зберігає поточну нотатку"""
    title = self.title_entry.get().strip()
    content = self.content_text.get('0.0', tk.END).strip()

    if not title:
        title = "Без назви"

    note_data = {
        'title': title,
        'content': content
    }

    if self.current_note is not None:
        # Оновлюємо існуючу нотатку
        self.notes[self.current_note] = note_data
    else:
        # Додаємо нову нотатку
        self.notes.append(note_data)

```

```

self.save_notes()
self.update_notes_list()
messagebox.showinfo("Успіх", "Нотатку збережено!")

```

```

def delete_note(self):
    """Видаляє поточну нотатку"""
    if self.current_note is not None:
        if messagebox.askyesno("Підтвердження", "Видалити нотатку?"):
            del self.notes[self.current_note]
            self.save_notes()
            self.update_notes_list()
            self.clear_form()
            self.current_note = None

```

```

def clear_form(self):
    """Очищає форму редагування"""
    self.title_entry.delete(0, tk.END)
    self.content_text.delete('0.0', tk.END)

```

```

def run(self):
    """Запускає додаток"""
    self.root.mainloop()

```

Запуск додатку

```

if __name__ == "__main__":
    app = NotesApp()
    app.run()

```

Порівняння Tkinter та CustomTkinter

Характеристика	Tkinter	CustomTkinter
Зовнішній вигляд	Класичний, застарілий	Сучасний, стильний
Теми	Обмежені	Світла/темна, легко змінюється
Віджети	Базові	Розширені, з

		округленими кутами
Складність	Простий	Дуже простий (аналогічний API)
Залежності	Вбудований	Потребує встановлення

Підсумки

1. **Менеджери компоновання** визначають, як віджети розташовуються у вікні:

- pack – для простих лінійних макетів
- grid – для складних табличних макетів
- place – для точного позиціонування

2. **Розширені віджети Tkinter** дозволяють створювати професійні інтерфейси:

- Listbox, Combobox для вибору
- Scale, Spinbox для регулювання
- Treeview для табличних даних
- Діалогові вікна для роботи з файлами

3. **CustomTkinter** – сучасна альтернатива з красивим дизайном:

- Підтримка тем (світла/темна)
- Стильні віджети
- Простий API, сумісний з Tkinter

4. **Практичне застосування:**

- Форми введення даних
- Панелі адміністрування
- Особисті менеджери (нотатки, завдання)
- Невеликі бізнес-додатки

Це завершує наш курс з програмування на Python. Ви пройшли шлях від абсолютного початківця до розробника, здатного створювати складні програми з графічним інтерфейсом. Бажаю успіхів у подальшому вивченні та застосуванні Python!

Питання для обговорення

1. Поясніть, що таке GUI і чому графічний інтерфейс є важливим для користувацьких програм. Які переваги мають програми з GUI порівняно з консольними?

2. Як створити головне вікно в Tkinter? Що таке цикл подій (mainloop()) і чому він необхідний? Що станеться, якщо його не викликати?

3. Напишіть програму з Tkinter, яка створює вікно з заголовком "Моя програма", розміром 400x300 пікселів, зеленим фоном та текстовою міткою "Привіт, світе!".

4. Поясніть різницю між трьома менеджерами компоновання в Tkinter: pack, grid та place. У яких ситуаціях доцільно використовувати кожен з них?

5. Проаналізуйте код:

```
import tkinter as tk
root = tk.Tk()

label1 = tk.Label(root, text="Ім'я:")
entry1 = tk.Entry(root)
button = tk.Button(root, text="Натисни")
```

```
label1.grid(row=0, column=0)
entry1.grid(row=0, column=1)
button.grid(row=1, column=0, columnspan=2)
```

Поясніть, як працює grid у цьому прикладі. Що означає columnspan=2?

6. Що таке події в Tkinter? Як прив'язати функцію до натискання кнопки? Напишіть програму з полем введення та кнопкою, при натисканні на яку текст з поля виводиться у мітці.

7. Як отримати текст з поля Entry? Як очистити поле введення? Як встановити текст у полі за замовчуванням?

8. Проаналізуйте код:

```
from tkinter import messagebox
result = messagebox.askyesno("Питання", "Ви впевнені?")
if result:
    print("Користувач підтвердив")
else:
```

ПІСЛЯМОВА

Шановні студенти! Ось ми й дійшли до фіналу нашого курсу. Ви подолали довгий шлях від першого знайомства з Python до створення власних графічних додатків. Це була насичена подорож, сповнена нових відкриттів, складних викликів і, сподіваюся, радості від перемог над черговою складною темою.

Що ми вивчили?

Ми розпочали з найосновнішого – зрозуміли, що таке алгоритми та програми, навчилися створювати змінні та працювати з різними типами даних. Поступово наші програми ставали складнішими: ми додали умовні конструкції, цикли, навчилися обробляти колекції даних. Функції допомогли нам структурувати код і уникати повторень, а модулі та пакети – організовувати великі проекти.

Особливу увагу ми приділили структурам даних – адже правильний вибір структури часто визначає ефективність усієї програми. Списки, кортежі, словники, множини, стеки, черги – кожна з цих структур має своє призначення і свої переваги. Розуміння алгоритмів сортування та пошуку дало нам змогу зазирнути "під капот" вбудованих функцій Python.

Об'єктно-орієнтоване програмування відкрило новий вимір у створенні програм. Класи та об'єкти, інкапсуляція, наслідування, поліморфізм – це не просто абстрактні поняття, а потужні інструменти для моделювання реального світу в коді. Магічні методи зробили наші класи "розумними" та інтегрованими з синтаксисом Python.

І нарешті, графічні бібліотеки Turtle та Matplotlib, а також створення GUI з Tkinter та CustomTkinter показали, як наші програми можуть взаємодіяти з користувачем через вікна, кнопки, поля введення та візуалізації.

Що далі?

Завершення цього курсу – не кінець, а лише початок вашого шляху як програміста. Ви отримали міцний фундамент, на якому можна будувати подальшу кар'єру. Ось кілька напрямків, у яких ви можете розвиватися:

- **Веб-розробка** – вивчіть фреймворки Django або Flask для створення веб-додатків
- **Data Science та Machine Learning** – заглибтеся в бібліотеки NumPy, Pandas, Matplotlib, Scikit-learn
- **Автоматизація** – створюйте скрипти для автоматизації рутинних завдань
- **Розробка ігор** – спробуйте бібліотеку Pygame

- **Мобільна розробка** – досліджуйте Kivy або BeeWare
- **Поглиблене ООП** – вивчайте патерни проєктування, чисту архітектуру, SOLID принципи

Найважливіші поради на майбутнє:

1. **Практикуйтеся щодня.** Програмування – це навичка, яка потребує постійного тренування. Навіть 15-30 хвилин щоденного кодування краще, ніж 5 годин раз на тиждень.

2. **Читайте чужий код.** Вивчайте відкриті проєкти на GitHub, аналізуйте, як досвідчені розробники структурують свої програми.

3. **Пишіть читабельний код.** Пам'ятайте, що код пишеться для людей, а лише потім для комп'ютерів. Дотримуйтеся PEP 8, давайте змістовні назви змінним, коментуйте складні місця.

4. **Навчіться користуватися системою контролю версій Git.** Це обов'язковий інструмент у сучасній розробці.

5. **Не бійтеся складних завдань.** Саме вони розвивають найбільше. Коли стикаєтеся з проблемою, розбивайте її на маленькі частини і розв'язуйте послідовно.

6. **Спілкуйтеся з колегами.** Беріть участь у форумах, відвідуйте мітапи, долучайтесь до спільнот. Обговорення кодів із іншими розробниками дуже збагачує.

Пам'ятайте, що всі великі програмісти колись були новачками. Кожен із них проходив через помилки, нерозуміння, розчарування – але вони не зупинялися. Наполегливість і цікавість – ось що відрізняє успішних розробників.

Дякую вам за цю спільну подорож. Сподіваюся, що матеріали курсу були для вас корисними, цікавими та надихаючими. Бажаю вам успіхів у подальшому вивченні програмування, цікавих проєктів і професійного зростання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Алгоритмізація і програмування : метод. реком. до виконання практичних робіт здобувачами вищої освіти ступеня «молодший бакалавр» факультету менеджменту спеціальності 122 «Комп'ютерні науки» денної форми навчання / уклад. Ю. В. Волосюк. Миколаїв : МНАУ, 2021. 52 с.
URL: <https://dspace.mnau.edu.ua/jspui/handle/123456789/10866>
2. Алгоритмізація та програмування. Частина 1. Базові концепції програмування. Лабораторний практикум : навчальний посібник / уклад. В. В. Романов, Т. І. Просянкіна-Жарова, О. Ю. Безносик. Київ : КПІ ім. Ігоря Сікорського, 2022. 150 с.
URL: <https://ela.kpi.ua/server/api/core/bitstreams/596d62b6-8251-4013-9c48-960e10b6dfb2/content>.
3. Бандоріна Л. М., Климкович Т. О., Удачина К. О. Основи алгоритмізації та програмування : навч. посібник. Дніпро : УДУНТ, 2022. 158 с.
URL: <https://crust.ust.edu.ua/server/api/core/bitstreams/bfb9f823-172b-49bf-92a2-48bbbef5f113/content>.
4. Беррі П. Head First Python. Легкий для сприйняття довідник. Харків : Фабула, 2023. 624 с.
5. Булгакова О. С., Зосімов В. В., Ходякова Г. В. Алгоритмізація і програмування: теорія та практика : навчальний посібник. Миколаїв : СПД Румянцева, 2021. 138 с.
URL: <http://dspace.mdu.edu.ua/jspui/handle/123456789/931>.
6. Васильєв О. М. Програмування мовою Java. Тернопіль : Богдан НК, 2022. 696 с.
7. Висоцька В. А., Оборська О. В. Python: алгоритмізація та програмування : навчальний посібник. Львів: Новий світ-2000, 2021. 514 с.
8. Григорович В. Г. Алгоритмізація та програмування. Частина 1 : навчальний посібник. Львів : Магнолія 2006, 2023. 357 с.
9. Злобін Г. Г. Основи алгоритмізації та програмування мовою Сі : підручник. Київ : Каравела, 2022. 168 с.
10. Кублій Л. І. Алгоритмізація та програмування. Практикум : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2019. 209 с. URL: <https://ela.kpi.ua/server/api/core/bitstreams/a2c179fd-fb1f-4536-aae7-08a8928f8569/content>
11. Лосєв М. Ю., Федорченко В. М. Програмування мовою Python : навчальний посібник. Харків; Львів : Новий Світ – 2000, 2024. 178 с.

12. Мартін Р. Чистий кодер. Кодекс поведінки для професійних розробників. Харків : Фабула, 2023. 256 с.
13. Пастернак І. І., Костик А. Т. Інструментальні засоби веб-технологій : навчальний посібник. Львів : Магнолія, 2024. 197 с.
14. Ришковець Ю. В., Висоцька В. А. Алгоритмізація та програмування. Частина 2 : навчальний посібник. Львів : Новий Світ – 2000, 2020. 320 с.
15. Рудий Т. В., Паранчук Я. С., Сенік В. В. Алгоритмізація та програмування. Частина 2. Модульне програмування : навчальний посібник. Львів : Львівський державний університет внутрішніх справ, 2024. 176 с. URL: <https://dspace.lvduvs.edu.ua/handle/1234567890/6994>
16. Рудий Т. В., Паранчук Я. С., Сенік В. В. Алгоритмізація та програмування. Частина 1. Структурне програмування : навчальний посібник. Львів : Львівський державний університет внутрішніх справ, 2023. 240 с. URL: <https://dspace.lvduvs.edu.ua/handle/1234567890/5515>
17. Селіверстов Р., Мельничин А. Основи програмування мовою Python : навчальний посібник. Львів : ЛНУ, 2020. 190 с.
18. Спирінцева О. В., Литвинов О. А., Герасимов В. В. Java-технології та мобільні пристрої. Алгоритми і структури даних : навч. посіб. Дніпро : ДНУ, 2016. 140 с.
19. Трофименко О. Г., Манаков С. Ю., Ларін Д. Г. Основи програмної інженерії : навч.-метод. посібник. Одеса : Фенікс, 2022. 197 с. URL: <https://dspace.onua.edu.ua/items/25698e60-3d50-42c0-a87a-27bd8b22a54d>
20. Трофименко О. Г., Прокоп Ю. В., Логінова Н. І., Задерейко О. В. C++. Алгоритмізація та програмування : підручник. Одеса : Фенікс, 2019. 477 с. URL: <https://dspace.onua.edu.ua/items/6c40c92b-c7d4-43ae-93dae195f3daf3d1>
21. Щербаков О. В., Парфьонов Ю. Е., Федорченко В. М. Основи об'єктно-орієнтованого програмування : навчальний посібник. Харків : ХНЕУ, 2019. 237с. URL: <http://surl.li/oespq>.

Навчальне видання

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Конспект лекцій

Укладачі: **Пархоменко** Олександр Юрійович
Тищенко Світлана Іванівна
Ємельянов Святослав Ігорович
Жебко Олександр Олегович
Богатєнкова Олександра Євгенівна

Формат 60x84 1/16. Ум. друк. арк. 4,0.
Тираж 20 прим. Зам. № _____

Надруковано у видавничому відділі
Миколаївського національного аграрного університету
54008, м. Миколаїв, вул. Георгія Гонгадзе, 9

Свідоцтво суб'єкта видавничої справи ДК № 4490 від 20.02.2013 р.