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

Миколаїв – 2026

УДК 004.42
А45

Друкується за рішенням науково-методичної комісії факультету менеджменту Миколаївського національного аграрного університету (протокол №8 від 23 квітня 2026 року)

Укладач:

О. Ю. Пархоменко – к.ф.-м.н., доцент 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., доцент, доцент кафедри кафедри електроенергетики, електротехніки та електромеханіки Миколаївського національного аграрного університету.

ЗМІСТ

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 TIOBE 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 запускайте 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 втілилося 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 читаєте.
- **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 (зрозуміємо print() глибше, з параметрами sep та end), отримувати дані від користувача через input() та навчимося форматувати 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. Давайте 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 отримаємо 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 начинок 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 ("A" < "a"). 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
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

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 (с посилається на той самий об’єкт, що й а)

# 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 escape-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("\nГарного 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 пройтися (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:27.2f} 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("📊 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:**

```
# Хочемо 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
Потрібно створити нескінченний цикл	while True
Очікування певної події (наприклад, введення правильного пароля)	while

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 × 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. Давайте 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)]  # списокове включення (list comprehension)
```

Про списокові 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("\nРОМБ:")

# 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(n²) 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(n²) – 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(n²)**.

```
# O(n) – лінійна складність
n = 1000
for i in range(n):
    print(i)  # 1000 операцій

# O(n²) – 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(n²) 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 <= ni < 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)?

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.
3. Бандоріна Л. М., Климкович Т. О., Удачина К. О. Основи алгоритмізації та програмування: навч. посібник. Дніпро: УДУНТ, 2022. 158 с.
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 для дистанційного навчання. Миколаїв: СПД Румянцева, 2021. 138 с.
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. для здобувачів ступеня бакалавра за спеціальністю 122 "Комп'ютерні науки" / ; КПІ ім. Ігоря Сікорського. Київ: КПІ ім. Ігоря Сікорського, 2019. 209 с.
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., Сеник В. В. Алгоритмізація та програмування. Частина 2. Модульне програмування: навчальний посібник. Львів: Львівський державний університет внутрішніх справ, 2024. 176 с.
16. Рудий Т. В., Паранчук Я. С., Сеник В. В. Алгоритмізація та програмування. Частина 1. Структурне програмування: навчальний посібник. Львів: Львівський державний університет внутрішніх справ, 2023. 240 с.
17. Селіверстов Р., Мельничин А. Основи програмування мовою Python: навчальний посібник. Львів: ЛНУ, 2020. 190 с.
18. Спірінцева 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.
20. Трофименко О. Г., Прокоп Ю. В., Логінова Н. І., Задерейко О. В. C++. Алгоритмізація та програмування: підручник. Одеса: Фенікс, 2019. 477 с.
21. Щербаков О. В., Парфьонов Ю. Е., Федорченко В. М. Основи об`єктно-орієнтованого програмування: навчальний посібник. Харків: ХНЕУ, 2019. 237с.

Навчальне видання

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

Укладач:

Пархоменко Олександр Юрійович

Формат 60x84 1/16. Ум. друк. арк. 4,8.
Тираж 5 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.