Міністерство освіти і науки України

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Конспект лекцій (частина друга)
для здобувачів першого (бакалаврського) рівня вищої освіти
освітньої програми «Інформаційні системи
та технології охорони і безпеки»
галузь знань F Інформаційні технології
спеціальності F6 Інформаційні системи та технології
денної та заочної форм навчання

Луцьк 2025

УДК 004.421 (07)

А 45

Рекомендовано до видання вченою радою факультету КІТ ЛНТУ,
протокол № _____ від « ____» _____ 20 ___ року.

Голова вченої ради факультету КІТ _____ Інна КОНДІУС

Електронна копія друкованого видання передана для внесення в репозитарій ЛНТУ

Директор бібліотеки _____ Наталія ПОЛІЩУК

Розглянуто і схвалено на засіданні кафедри комп'ютерної інженерії та безпеки
ЛНТУ, протокол № _____ від « ____» _____ 20 ____ року.

Завідувач кафедри КІБ _____ Тарас ТЕРЛЕЦЬКИЙ

Укладач: _____ Світлана ЛАВРЕНЧУК, кандидат технічних наук, доцент кафедри комп'ютерної інженерії та безпеки ЛНТУ

Рецензент: _____ Петро ПЕХ, кандидат технічних наук,
доцент кафедри комп'ютерної інженерії та безпеки ЛНТУ

Відповідальний за випуск: _____ Тарас ТЕРЛЕЦЬКИЙ, кандидат технічних наук, доцент кафедри комп'ютерної інженерії та безпеки ЛНТУ

А 45

Алгоритмізація та програмування: конспект лекцій (частина друга) для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Інформаційні системи та технології охорони і безпеки» галузь знань F Інформаційні технології спеціальності F6 Інформаційні системи та технології денної та заочної форм навчання / уклад. С. В. Лавренчук. Луцьк: ЛНТУ, 2025. 248 с.

Конспект лекцій (частина друга) з дисципліни «Алгоритмізація та програмування» складений відповідно до діючої програми курсу.

Призначений для здобувачів вищої освіти спеціальності Інформаційні системи та технології освітньої програми «Інформаційні системи та технології охорони і безпеки».

ЗМІСТ

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням обсягів даних, складністю інформаційних систем та підвищеними вимогами до їхньої надійності, ефективності й безпеки. У цих умовах особливого значення набувають фундаментальні знання з алгоритмізації та програмування, що формують базу для подальшої професійної підготовки фахівців у галузі інформаційних систем та технологій охорони і безпеки.

Метою вивчення дисципліни *«Алгоритмізація та програмування»* є формування у здобувачів освіти системних знань про принципи побудови алгоритмів, основні елементи структурного програмування та практичні навички реалізації алгоритмів.

Друга частина конспекту лекцій присвячена вивченню мови програмування Python, яка сьогодні є однією з найпопулярніших і найгнучкіших у галузі інформаційних технологій. Простота синтаксису, розвинена бібліотечна екосистема, підтримка різних парадигм програмування та потужні інструменти обробки даних роблять Python універсальним інструментом для розв’язання задач у сфері безпеки, аналітики даних, штучного інтелекту та автоматизації.

Мета даного навчального матеріалу – сформувати у здобувачів освіти базові знання та практичні навички програмування мовою Python, навчити застосовувати основні алгоритмічні конструкції, структури даних і принципи модульного програмування для створення ефективних програмних рішень.

В цьому методичному виданні розглядаються такі питання:

– базові типи даних і оператори Python, принципи введення та виведення інформації;

– умовні конструкції, цикли та спискові структури;

– функції, кортежі, словники, модулі та пакети;

– методи роботи з текстовими даними та файлами;

– підходи до розроблення алгоритмів і оцінки їх складності.

Матеріали конспекту лекцій містять систематизований теоретичний виклад основних тем курсу, приклади програмного коду та контрольні питання.

ТЕМА 1. Типи даних та оператори в Python, функції вводу-виводу

1.1 Загальні відомості про мову Python

1.1.1 Історія розвитку та версії мови

Python є *інтерпретованою мовою*. Якщо ми хочемо програмувати на Python, нам знадобиться інтерпретатор Python. Без нього ми не зможемо запустити свій код. На щастя, Python безкоштовний. В силу історичних причин, мови, призначені для використання в режимі інтерпретації, часто називають *скриптовими мовами*, а вихідні програми, написані з їх допомогою, називаються *скриптами*.

Отож, *Python* – широко розповсюджена, інтерпретована, об'єктно-орієнтована мова програмування високого рівня з динамічною семантикою, що використовується для програмування загального призначення.

Однією з дивовижних особливостей Python є те, що це фактично робота однієї людини. Python був створений Гвідо ван Россум який народився в 1956 році в Гарлемі, Нідерланди. Швидкість, з якою Python поширився по всьому світу, є результатом безперервної роботи тисяч (дуже часто анонімних) програмістів, тестувальників, користувачів (багато з них не є IT-фахівцями) та ентузіастів, але слід сказати, що саме перша ідея (зерно, з якого проріс Python) прийшла в голову одній людині – Гвідо.

У 1999 році Гвідо ван Россум визначив для себе цілі створення Python:

– легка та інтуїтивно зрозуміла мова, така ж потужна, як і в основних конкурентів;

– відкритий вихідний код, щоб будь-хто міг зробити свій внесок у його розвиток;

– код зрозумілий, як і звичайна англійська мова;

– підходить для повсякденних завдань, що дозволяє скоротити час розробки.

Python має двох прямих конкурентів із схожими властивостями та схильностями. А саме:

– Perl – скриптова мова програмування, автором якої є Ларрі Уолл;

– Ruby – скриптова мова програмування, автором якої є Юкіхіро Мацумото.

Перша є більш традиційною і більш консервативною, ніж Python, і нагадує деякі старі мови, що походять від класичної мови програмування C.

На відміну від Python, остання є більш інноваційною та більш насиченою свіжими ідеями. Сам Python знаходиться десь між цими двома творіннями.

Існує дві основні версії Python - це Python 2 і Python 3.

Python 2 – це старіша версія оригінального Python. Зараз його розвиток навмисно зупинено, хоча це не означає, що для нього немає оновлень. Навпаки, оновлення випускаються на регулярній основі, але вони не мають на меті суттєво змінити мову. Вони радше виправляють будь-які щойно виявлені помилки та діри в безпеці. Шлях розробки Python 2 вже зайшов у глухий кут, але сам Python 2 все ще використовується.

Python 3 – це новіша (а точніше, поточна) версія мови. Вона проходить свій еволюційний шлях, створюючи власні стандарти та особливості.

Ці дві версії Python не сумісні між собою. Скрипти на Python 2 не будуть виконуватися в середовищі Python 3 і навпаки, тому якщо ви захочете, щоб старий код на Python 2 виконувався інтерпретатором Python 3, то єдиним можливим рішенням буде його переписування, звичайно, не з нуля, оскільки велику частину коду можна залишити недоторканою, але доведеться переглянути весь код, щоб знайти всі можливі несумісності. На жаль, цей процес неможливо повністю автоматизувати.

Python 3 – це не просто краща версія Python 2 – це зовсім інша мова, хоча й дуже схожа на свою попередницю. Крім Python 2 і Python 3, існує декілька версій кожної з них.

Класична реалізація Python під назвою CPython є еталонною версією комп’ютерної мови Python Гвідо ван Россума, і саме її найчастіше називають «Python». Коли ви чуєте назву CPython, швидше за все, вона використовується для того, щоб відрізнити її від інших, нетрадиційних, альтернативних реалізацій.

Існують Python, які супроводжується людьми, зібраними навколо PSF (Python Software Foundation), спільноти, яка прагне розвивати, покращувати,

розширювати та популяризувати Python та його середовище. Президентом PSF є сам Гвідо фон Россум, і саме тому ці «Python» називаються *канонічними*. Вони також вважаються *еталонними Python*, оскільки будь-яка інша реалізація мови має відповідати всім стандартам, встановленим PSF.

Гвідо ван Россум використав мову програмування «C» для реалізації найпершої версії своєї мови, і це рішення залишається чинним досі. Усі Python, що надходять із PSF, написані на мові «C». Для такого підходу є багато причин. Одна з них (мабуть, найважливіша) полягає в тому, що завдяки їй Python може бути легко портований і перенесений на всі платформи з можливістю компіляції та виконання програм мовою «C» (практично всі платформи мають цю функцію, що зберігає для Python багато нових можливостей для розвитку).

Саме тому реалізацію PSF часто називають *CPython*. Це найвпливовіший Python серед усіх Python у світі.

Python – багатоцільова мова програмування, яка дозволяє писати код, що добре читається.

Відносний лаконізм мови Python дозволяє створити програму, яка буде набагато коротше свого аналога, написаного на іншій мові.

Python – багатоплатформова мова програмування. Це означає, що програми на Python можна запускати в різних операційних системах без будь-яких змін. Ще однією перевагою Python є його стандартна бібліотека, яка встановлюється разом з Python і містить готові інструменти для роботи з операційною системою, веб-сторінками, базами даних, різними форматами даних, для побудови графічного інтерфейсу програм тощо.

1.1.2 Переваги та недоліки Python

Переваги Python:

1. Це мова програмування, що інтерпретується: програма мовою Python запускається прямо з вихідного коду.

2. Це високорівнева мова програмування.

3. Це платформонезалежна мова:

– програми на Python можна створювати різних операційних системах (Linux, Windows, OS X);

– програми на Python можна запускати різних операційних системах (Linux, Windows, OS X).

4. Це open source проект.

5. Це проста мова.

6. Це вбудовувана скриптова мова.

7. Це динамічна мова, що спрощує написання нескладних програм.

8. Для Python існує велика бібліотека класів на будь-який смак.

Недоліки Python:

1. Низька швидкість виконання порівняно з такими мовами, як C та C++.

2. Динамічна типізація мови – мінус при написанні складних програм.

1.2 Завдання, які вирішуються за допомогою Python

Python підходить для вирішення широкого спектра завдань. Розіб'ємо їх на категорії:

1. Системне програмування. Вбудовані в Python інтерфейси доступу до служб операційних систем роблять його ідеальним інструментом для створення програм, що переносяться, і утиліт системного адміністрування.

2. Графічні програми. Простота Python і швидкість розробки роблять його чудовим засобом створення графічного інтерфейсу. До складу Python входить стандартний об'єктно-орієнтований інтерфейс до GUI API.

3. Веб-програми. За допомогою додаткових фреймворків мовою Python (Django, Flask, Pyramid) можна створювати повнофункціональні сайти.

4. Веб-сценарії. Python поставляється разом із стандартними інтернет-модулями, які дозволяють програмам виконувати різноманітні мережеві операції як у режимі клієнта, так і в режимі сервера.

5. Інтеграція компонентів. Можливість Python розширюватися та вбудовуватися в системи мовою C++ робить його зручним для опису поведінки інших систем та компонентів.

6. Програми баз даних. У Python є інтерфейси доступу до всіх основних реляційних баз даних: Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL, SQLite та багато інших. З їх допомогою можна створювати програми баз даних.

1.3 Практичні приклади використання Python

Практичні приклади використання Python:

1. Компанія Google використовує Python у своїй пошуковій системі. На Python пишуть в Google, Facebook, Instagram, Spotify, Netflix.

2. Компанії Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm та IBM використовують Python для тестування апаратного забезпечення.

3. Сервіс YouTube значною мірою реалізований на Python.

4. Агентство національної безпеки (NSA) використовує Python для шифрування та аналізу даних.

5. Компанії JPMorgan Chase, UBS, Getco та Citadel застосовують Python для прогнозування фінансового ринку.

6. Програма BitTorrent для обміну файлами в пірінгових мережах написана мовою Python.

7. NASA, Los Alamos, JPL та Fermilab використовують Python для наукових обчислень.

8. Веб-основа Spotify, що складається з багатьох взаємопов’язаних сервісів, значною мірою спирається на Python.

9. Відомо, що OpenAI, провідна дослідницька лабораторія AI, використовує фреймворк Python PyTorch як стандарт для глибокого навчання та тренування своїх AI-систем. Google також активно використовує Python у своїх проєктах з AI, ML та робототехніки.

10. Amazon використовує Python для автоматизації.

11. The Sims 4, World of Tanks, EVE Online і Battlefield 2 — це лише деякі з найбільш відомих ігор, які використовують Python.

12. Більшість сервісів Uber працюють на Python / Node.js.

13. Dropbox – хмарна платформа, призначена для зберігання та обміну файлами між пристроями. Спочатку проект мав лише 2000 користувачів, а сьогодні ним користуються понад 200 мільйонів людей. Dropbox використовує Python для крос-платформної підтримки та швидкого циклу розробки програм.

14. Найпопулярніші бібліотеки Python для веб-скрапінгу – це Beautiful Soup і Selenium. Однак повний набір для вебскрапінгу також містить:

– Lxml module – потужна бібліотека для обробки XML та HTML документів.

– Urllib module – вбудована бібліотека з функціями для роботи з URL-адресами.

– PyautoGUI – кросплатформна бібліотека автоматизації GUI, яка дозволяє керувати мишею та клавіатурою для автоматизації завдань.

– Schedule – проста бібліотека, яка допомагає зазначати інтервали виконання функцій Python.

1.4 Функція print()

1.4.1 Синтаксис функції

Найпростіша програма мовою Python може виглядати так, як на рисунку 1.1.

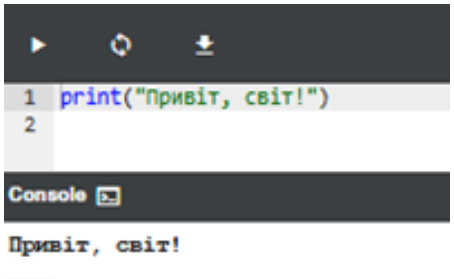


Рисунок 1.1 – Найпростіша програма

Як бачимо з рисунку, вона просто виводить повідомлення на екран. Розглянемо її складові:

– слово `print`;

– відкриваюча дужка;

– лапки відкриваються;

– рядок тексту: *Привіт, світ!*;

– лапки закриваються;

– закриваюча дужка.

Все, що перерахованого вище, відіграє дуже важливу роль у коді.

Слово `print`, яке ви тут бачите, є назвою функції. Це не означає, що де б це слово не зустрічалося, воно завжди означає назву функції. Значення слова випливає з контексту, в якому воно було вжито.

Ви, напевно, багато разів зустрічали термін функція на уроках математики. Напевно, ви також можете назвати кілька назв математичних функцій, таких як синус або логарифм.

Функції Python, натомість, є більш гнучкими і можуть мати більше можливостей, ніж їхні математичні родичі.

Функція (в даному контексті) – це окрема частина комп'ютерного коду, яка здатна виконувати певні дії:

– викликати певний результат (наприклад, відправити текст на термінал, створити файл, намалювати зображення, відтворити звук і т.д.); цього немає у світі математики;

– обчислювати значення (наприклад, квадратний корінь числа або довжину заданого тексту) і повертати його як результат; виконання функції – це те, що робить функції Python близькими до математичних понять.

Більше того, багато функцій Python можуть робити ці дві речі одночасно.

Функції можуть бути:

– *вбудовані* в сам Python. Функція print є однією з таких; така функція є додатковою цінністю, отриманою разом із Python та його середовищем (вона *вбудована*); вам не потрібно робити нічого особливого (наприклад, просити будь-кого про що-небудь), якщо ви хочете нею скористатися. Перелік вбудованих функцій Python можна знайти за посиланням: https://docs.python.org/3/library/functions.html ;

– вони можуть бути вбудовані в одне або кілька доповнень Python, так звані *модулі*; деякі модулі постачаються разом із Python, інші можуть вимагати

окремої інсталяції – у будь-якому випадку, усі вони мають бути явно пов’язані з вашим кодом (розглянемо пізніше як це);

–ви можете *написати їх самостійно*, розмістивши у вашій програмі стільки функцій, скільки забажаєте або буде потрібно, щоб зробити її простішою, зрозумілішою та елегантнішою.

1.4.2 Аргументи функції

Математичні функції зазвичай приймають один аргумент. Наприклад, `sin(x)` отримує аргумент `x`, який є величиною кута.

Функції Python більш універсальні. Залежно від потреби, вони можуть приймати будь-яку кількість аргументів – стільки, скільки необхідно для виконання своїх задач. Примітка: може бути нуль аргументів, тобто деякі функції Python не потребують жодного аргументу.

Незважаючи на кількість необхідних/наданих аргументів, функції Python *обов’язково потребують наявності пари круглих дужок* – відкриваючої та закриваючої відповідно. Якщо ви хочете передати функції один або кілька аргументів, ви розміщуєте їх у круглих дужках. Якщо ви збираєтеся використовувати функцію, яка не приймає жодного аргументу, ви все одно повинні поставити круглі дужки, але залишити їх порожніми.

Наявність круглих дужок біля назви функції дає змогу також відрізняти звичайні слова від назв функцій. Це стандартна домовленість.

У прикладі з рисунку 1.1 єдиним аргументом, що передається у функцію `print()` є рядок, який обмежений лапками – власне, лапки і створюють рядок – вони вирізають частину коду і надають йому інше значення.

Тобто все, що вкладено в лапках, – не код, ця частина програми не призначена для виконання, він буде просто використаний в тому вигляді, як подано в лапках. Майже все, що ви помістите в лапки, буде сприйматися буквально, не як код, а як дані.

1.4.3 Використання кількох аргументів

Спробуймо передати у функцію `print()` більше одного аргументу (рисунок 1.2).

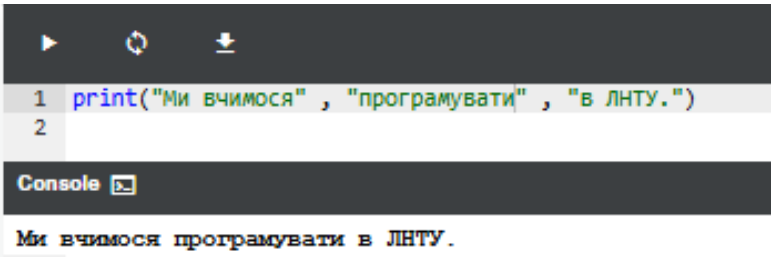


Рисунок 1.2 – Функція `print()` з трьома аргументами

Маємо один виклик функції `print`(), але він містить *три аргументи*. Всі вони – рядки.

Аргументи *відокремлюються комами*. Ми оточили їх пробілами, щоб зробити їх більш помітними, але це не є обов'язковим.

У цьому випадку коми, що розділяють аргументи, відіграють зовсім іншу роль, ніж кома всередині рядка. Перша є частиною синтаксису мови Python, а друга призначена для відображення в консолі.

Якщо ви подивитеся на код ще раз, то побачите, що всередині рядків коду (тих, що в лапках) немає пробілів.

З цього прикладу можна зробити два висновки:

– функція print(), викликана з більш ніж одним аргументом, виводить їх усі в один рядок;

– функція print() за власною ініціативою ставить пробіл між виведеними аргументами.

Спосіб, яким ми передаємо аргументи у функцію `print`(), є найпоширенішим у Python і називається *позиційним*. Ця назва походить від того, що значення аргументу визначається його позицією (наприклад, другий аргумент повинен виводитися після першого, а не навпаки) (рисунок 1.3).

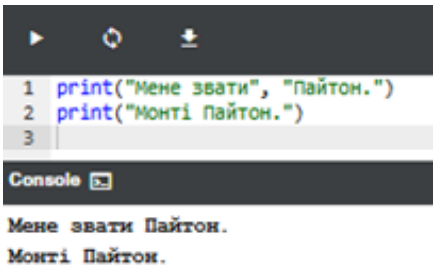


Рисунок 1.3 – Позиція аргументів у функції `print()`

1.4.4 Аргументи ключових слів

Функція `print()` має два аргументи ключових слів, які ви можете застосовувати для своїх цілей. Перший називається `end`.

У вікні редактора можна побачити дуже простий приклад використання аргументу ключового слова (рисунок 1.4).

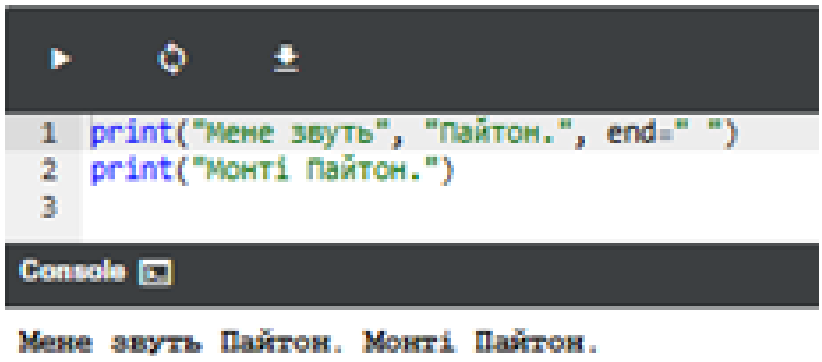


Рисунок 1.4 – Приклад використання аргументу `end`

Аргумент ключового слова складається з трьох елементів: ключового слова, що ідентифікує аргумент (в цьому випадку `end`); знаку рівності (=); і значення, що присвоюється аргументу (тут пробіл);

Будь-які аргументи ключового слова мають бути *розміщені після останнього позиційного аргументу* (це дуже важливо).

У нашому прикладі ми використали ключове слово `end` і задали його як рядок, що містить один пробіл.

Як бачимо, ключове слово `end` визначає символи, які функція `print()` посилає на виведення після того, як досягає кінця своїх позиційних аргументів.

Поведінка за замовчуванням описує ситуацію, коли аргумент ключового слова `end` використовується *неявно* наступним чином: `end="\n".`

Спробуймо забрати пробіл в значенні, що присвоюється аргументу `end`, тобто присвоєний йому рядок стане порожнім (рисунок 1.5).

Оскільки `end` аргумент був заданий порожнім, функція `print()` також нічого не виводить після того, як закінчаться її позиційні аргументи, тому відбудеться склеювання рядків.

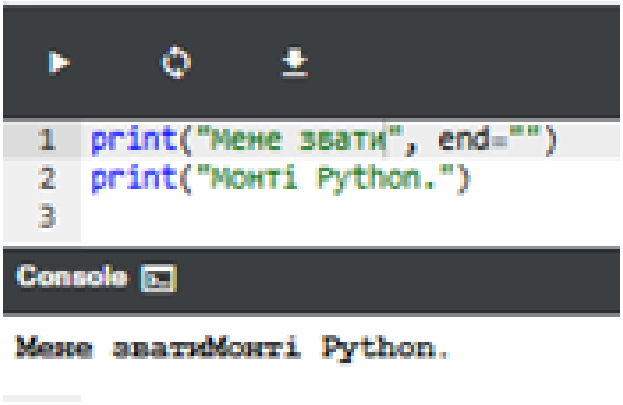


Рисунок 1.5 – Приклад використання аргументу `end` з порожнім присвоєнням

Рядок, що присвоюється аргументу ключового слова `end`, може мати довільну довжину.

Раніше ми говорили, що функція `print()` розділяє аргументи, що виводяться, пробілами. Цю поведінку також можна змінити.

Аргумент ключового слова, який може це зробити, називається `sep` (як роздільник, від англ. *separator*). Розглянемо приклад на рисунку 1.6.

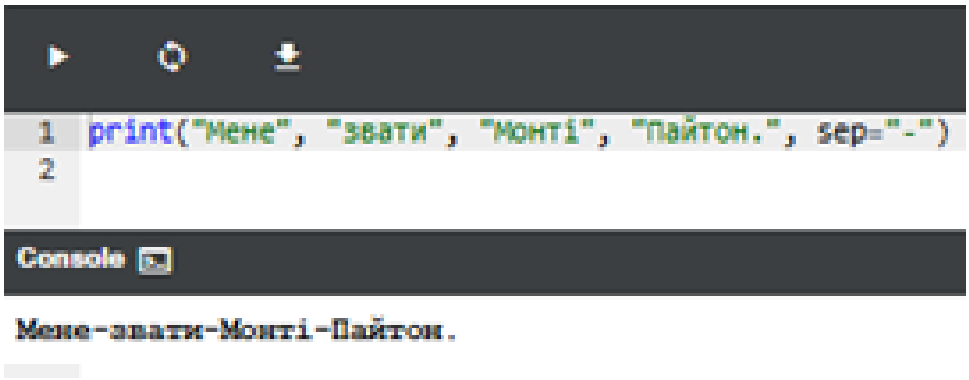


Рисунок 1.6 – Приклад заміни розділювача між словами

Функція *print()* тепер використовує тире замість пробілу для розділення виведених аргументів. Примітка: значенням аргументу *sep* може бути і порожній рядок. Тоді всі слова будуть «склеєні» між собою.

Обидва аргументи ключового слова можуть бути *поєднані в одному виклику*, як показано на рисунку 1.7.

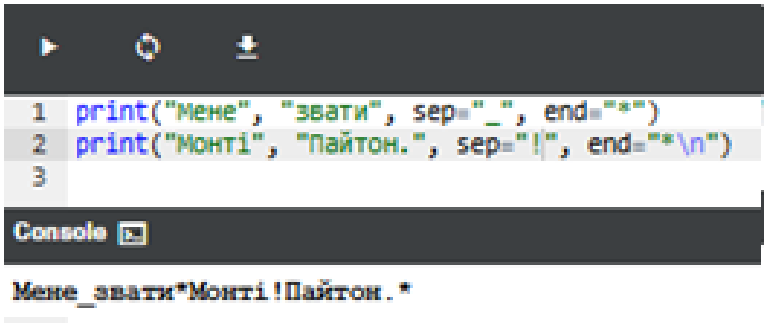


Рисунок 1.7 – Приклад використання обох аргументів ключового слова

1.4.5 Семантика функції

Ім’я функції (в даному випадку `print`) разом з круглими дужками та аргументом (аргументами) формує `виклик функції`.

Розглянемо, що відбувається, коли Python зустрічає виклик, наведений нижче:

```
назва_функції(аргумент)
```

– по-перше Python перевіряє, чи вказане ім’я є *легальним* (переглядає свої внутрішні дані (словники), щоб знайти наявну функцію з таким ім’ям; якщо цей пошук не вдається, Python перериває код);

– по-друге, Python перевіряє, чи *вимоги функції до кількості аргументів* дозволяють викликати функцію саме таким чином (наприклад, якщо конкретна функція вимагає рівно два аргументи, то будь-який виклик з передачею лише одного аргументу буде вважатися помилковим і призведе до переривання виконання коду);

– по-третє, Python на мить виходить з вашого коду і переходить до функції, яку ви викликаєте; звісно, він *приймає* ваш аргумент (аргументи) і *передає* його (їх) до *функції*;

– по-четверте, функція *виконує свій код*, досягається бажаний ефект (якщо такий є), оцінюється бажаний результат (результати) (якщо такі є) і завершує виконання поставленого перед нею завдання;

– нарешті, Python *повертається до вашого коду* (на місце одразу після виклику) і продовжує його виконання.

1.4.6 Функція print() та результат її виконання, аргументи та значення, які вона повертає

Функція *print()*:

– *приймає* аргументи (може приймати більше одного аргументу, а також може приймати менше одного аргументу, тобто 0);

– при необхідності *перетворює* їх у читабельну для людини форму (як ви можете здогадатися, рядки не потребують цієї дії, оскільки рядок і так є читабельним);

– *відправляє* отримані дані *на пристрій виводу* (як правило, консоль); іншими словами, все що ви передаєте у функцію print(), з'явиться на вашому екрані.

Функція `print()` може працювати практично з усіма типами даних, які пропонує Python. Рядки, числа, символи, логічні значення, об'єкти – все це можна успішно передавати в print().

Функція `print()` не повертає жодних значень (як, наприклад, функція sin(x)в математиці повертає результат обчислення).Основне призначення `print()` – вивід даних, а не їх обробка.

1.4.7 Інструкції Python

Ви вже бачили комп'ютерну програму, яка складається з виклику однієї функції. Виклик функції є одним з багатьох можливих типів інструкцій Python.

Звичайно, будь-яка складна програма містить набагато більше інструкцій, ніж одну. На відміну від більшості мов програмування, Python вимагає, щоб *у рядку* було *не більше однієї інструкції*.

Рядок може бути порожнім (тобто не містити жодної інструкції), але він не повинен містити двох, трьох або більше інструкцій. Це категорично заборонено.

Примітка. В Python є один виняток з цього правила – він дозволяє одній інструкції займати більше одного рядка (що може бути корисно, якщо ваш код містить складні конструкції).

Розглянемо код, що наведено на рисунку 1.8.

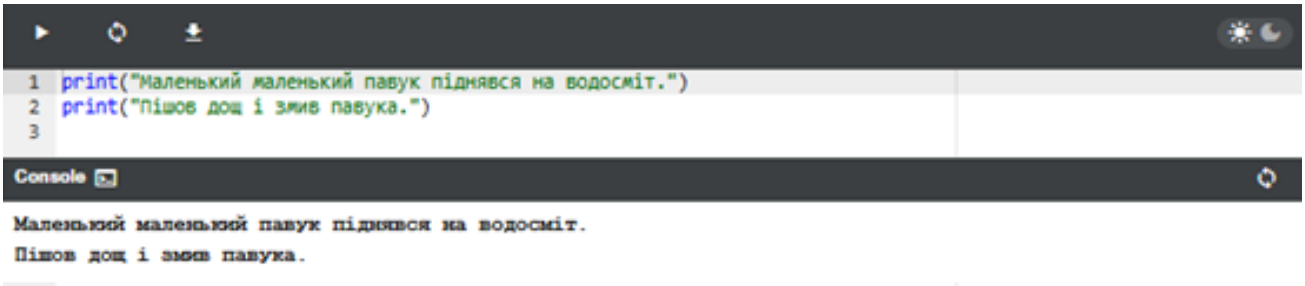


Рисунок 1.8 – Програма, що містить дві інструкції

Програма викликає функцію `print()` двічі, і в консолі можна побачити два окремі рядки – це означає, що функція `print()` починає свій *вивід з нового рядка* кожного разу, коли починає своє виконання; таку поведінку можна змінити, але можна і використати її на свою користь;

Кожен виклик `print()` містить в якості аргументу окремий рядок, а вміст консолі відображає його – це означає, що інструкції в коді виконуються в тому ж порядку, в якому вони були розміщені у вихідному файлі; жодна наступна інструкція не виконується до тих пір, поки не буде завершена попередня (з цього правила є деякі виключення, але їх можна поки що ігнорувати).

Розглянемо код, що наведено на рисунку 1.9.

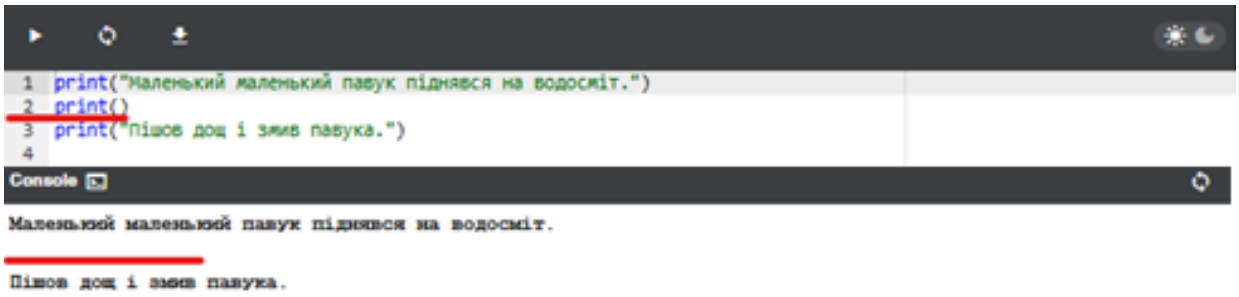


Рисунок 1.9 – Програма, що містить три інструкції

Як бачимо, порожній виклик `print()` не такий порожній, як можна було очікувати – він дійсно виводить порожній рядок, або (така інтерпретація також є правильною) виводить новий рядок.

Це не єдиний спосіб створення нового рядка в консолі виведення.

1.5 Екрануючий символ в Python та символ нового рядка

Змінимо програму, додавши комбінацію символів \n всередині речень (рисунок 1.10).

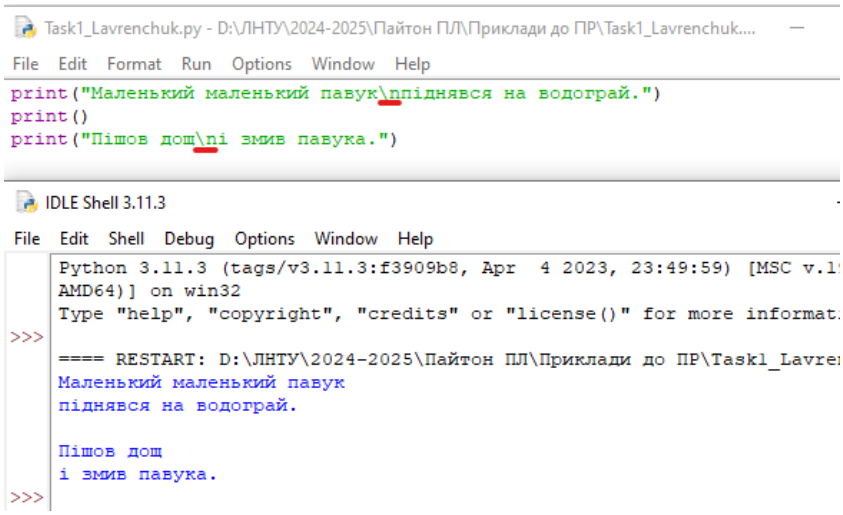


Рисунок 1.10 – Використання символу переходу на новий рядок

Як бачите, у тексті з’являються два нових рядки в тих місцях, де було застосовано \n. Хоча людина бачить два символи, Python бачить один.

Зворотний слеш (\) має спеціальне значення при використанні всередині рядків – він називається *символом екранування*.

Слово *екранування* слід розуміти конкретно – це значить, що ряд символів у рядку екранується на мить (дуже коротку мить), щоб здійснити спеціальну вставку.

Іншими словами, зворотний слеш сам по собі нічого не означає, а є лише своєрідним повідомленням про те, що наступний за ним символ також має інший вигляд і значення.

Літера n, що ставиться після зворотного слеша, походить від слова `newline` (новий рядок).

Зворотний слеш і n разом утворюють спеціальний символ, який називається символом переходу на новий рядок, що вказує консолі почати новий рядок для виведення.

Якщо ви хочете поставити лише один зворотний слеш всередині рядка, не забувайте про його екранування – його потрібно подвоїти. Наприклад, такий вираз призведе до помилки:

```
print("\")
```

а цей – ні:

```
print("\\")
```

Не всі екрановані пари (зворотний слеш у поєднанні з іншим символом) щось означають.

Ви отримаєте помилку, якщо ви використовуєте подвійні лапки всередині рядка, який оточують подвійні лапки (рисунок 1.11).

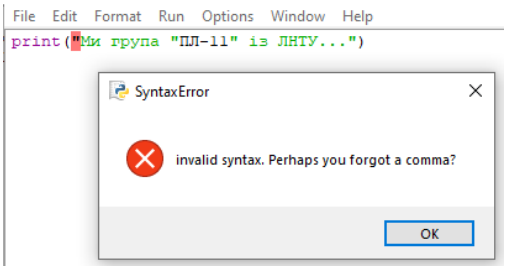


Рисунок 1.11 – Помилкове використання лапок без екранування

Щоб виправити цю проблему, використовують символ екранування \ (рисунок 1.12).

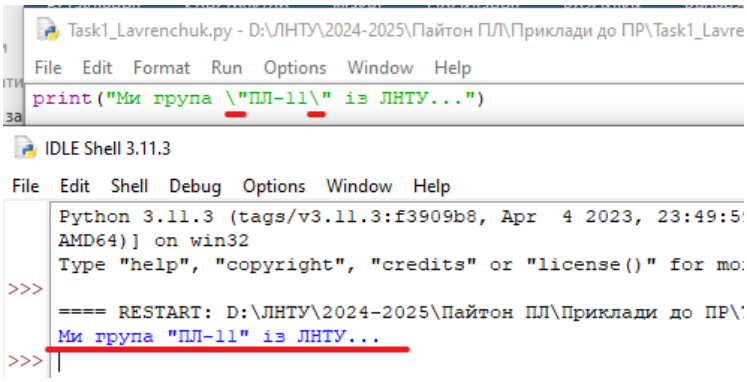


Рисунок 1.12 – Використання лапок з екрануванням

Інші символи екранування, що використовуються в Python, наведено в таблиці 1.1.

Таблиця 1.1 – Символи екранування

Код	**Результат**
\’	одинарні лапки всередині стрічки з одинарними лапками
\»	подвійні лапки всередині строки з подвійними лапками
\n	новий рядок
\\	зворотний слеш
\r	початок рядка (працює лише для Windows)
\t	табуляція (відступ, аналог натискання клавіші Tab)
\b	повернення на одну позицію (аналог Backspace на клавіатурі)
\f	форма подачі
\ooo	вісімковий еквівалент
\xhh	шістнадцятковий еквівалент
\a	Звуковий сигнал

1.6 Типи даних

1.6.1 Цілі числа

Розглянемо приклад (рисунок 1.13).

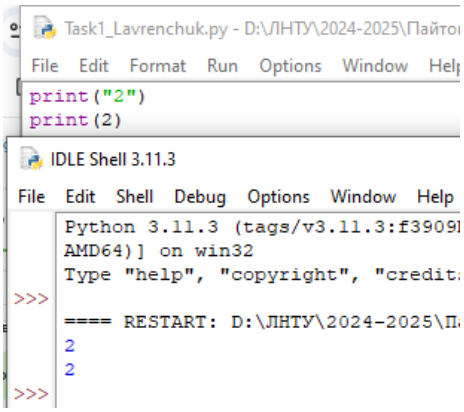


Рисунок 1.13 – Виведення різних типів даних

В результаті виводу ми бачимо два однакових рядки, але у цьому прикладі наведено два різних типи літералів: *рядок* і *ціле число*.

Функція `print()` виводить їх абсолютно однаково – цей приклад очевидний, оскільки їхнє зрозуміле для людини представлення також однакове. Всередині пам’яті комп’ютера, ці два значення зберігаються абсолютно по-різному – рядок існує як просто рядок (послідовність літер). Число перетворюється в машинне представлення (набір бітів). Функція `print()` здатна відобразити їх обидва у читабельному для людини вигляді.

Числа, з якими працюють сучасні комп’ютери, бувають двох типів:

– цілі числа, тобто ті, що позбавлені дробової частини;

– та числа з рухомою крапкою, що мають (або можуть мати) дробову частину (в математиці це «дійсні числа»).

Властивість числового значення, яка визначає його вид, діапазон і призначення, називається *типом*. Якщо ви кодуєте літерал і розміщуєте його всередині коду Python, то форма літералу визначається його представленням (типом), яке Python буде використовувати для його *зберігання в пам'яті*.

Ціле число можна записати або так: 11111111, або так: 11_111_111. Від'ємні числа можна записати: -11111111 або -11_111_111.

Примітка. У Python 3.6 додано підкреслення в числових літералах, що дозволяє ставити одинарні підкреслення між цифрами та після базових специфікаторів для покращення читабельності. Ця функція недоступна в старих версіях Python.

Перед додатними числами не обов'язково ставити знак плюс, але це допустимо, якщо ви вважаєте за потрібне це зробити. Наступні рядки описують одне і те саме число: +11111111 та 11111111.

Якщо перед цілим числом стоїть 0o або 0O (нуль-о), воно розглядатиметься як вісімкове значення. Це означає, що число має містити лише цифри з діапазону [0..7].

Наприклад, 0o123 – *вісімкове* число з відповідним десятковим значенням 83 (рисунок 1.14), а 0x123 – *шістнадцяткове* число з відповідним десятковим значенням 291 (рисунок 1.15).

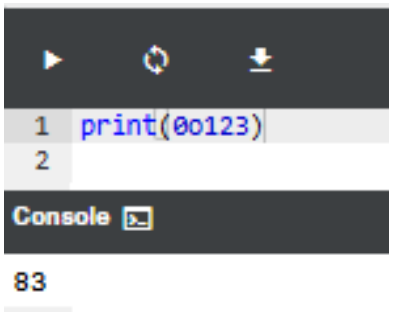


Рисунок 1.14 – Виведення вісімкового числа в 10-му вигляді

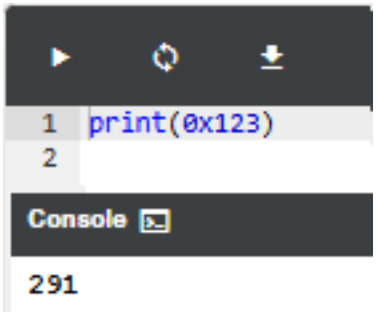


Рисунок 1.15 – Виведення шістнадцяткового числа в 10-му вигляді

Як бачимо з рисунків 1.14-1.15, функція `print()` виконує перетворення автоматично.

1.6.2 Дійсні числа

Кожного разу, коли ми використовуємо такі терміни, як два з половиною (`2.5`) або мінус нуль цілих чотири десятих (`-0.4`), ми думаємо про числа, які комп'ютер сприймає як числа з рухомою крапкою. *Зверніть увагу ще раз: між 2 і 5 стоїть крапка, а не кома.*

Значення нуль цілих чотири десятих можна записати на мові Python так `0.4`, але є просте правило – *нуль можна опускати, коли він є єдиною цифрою перед десятковою крапкою або після неї.*

По суті, можна записати значення 0.4 так: .4

Значення 4.0 можна записати так: 4.

Це не змінить ні його тип, ні значення.

Подивіться на ці два числа: 4 4.0

Вам може здатися, що вони абсолютно однакові, але Python бачить їх зовсім по-різному.

4 є *цілим* числом, тоді як 4.0 – число з *рухомою крапкою*.

Крапка це те, що відокремлює цілу частину від дробової.

З іншого боку, не тільки крапки визначають числа з рухомою крапкою. Ви також можете використовувати літеру e.

Для запису будь-яких дуже великих або дуже маленьких чисел можна використовувати наукові позначення.

Візьмемо, наприклад, швидкість світла, виражену в метрах за секунду. Прямим текстом це виглядатиме так: 300000000.

Щоб не писати стільки нулів, у підручниках з фізики використовують скорочену форму, яку ви, напевно, вже бачили: 3×10^8.

Вона читається так: три на десять в восьмому степені.

У Python той самий ефект досягається дещо іншим способом: 3E8.

Буква E (можна також використовувати малу літеру e – вона походить від слова експонента) – це стислий запис фрази, помноженої на десять у степені.

Примітка:

– показник степеня (значення після E) має бути цілим числом;

– основа (значення до символу E) може бути як цілим числом, так і числом з рухомою крапкою.

Іноді Python може використовувати *іншу систему позначень*, ніж ви.

Наприклад, припустимо, ви вирішили використати наступний літерал з рухомою крапкою: 0.000000000000000000001.

Коли ви пропускаєте цей літерал через Python:

```
print(0.000000000000000000001)
```

то отримуєте такий результат:

```
1e-22
```

Python завжди вибирає більш компактну форму представлення числа, і це слід враховувати при створенні літералів.

1.6.3 Рядки

Рядки використовуються, коли потрібно обробити текст (наприклад, імена всіх видів, адреси, словники тощо), а не числа.

Ви вже трохи знаєте про них, наприклад, що *рядки потребують лапок* так само, як числа з рухомою крапкою потребують крапок.

Це дуже типовий рядок: «Я студент».

Однак проблема полягає у тому, як закодувати лапки всередині рядка, який вже обрамлений лапками.

Припустімо, що ми хочемо надрукувати дуже просте повідомлення:

Мені подобається «ЛНТУ»

Як це зробити, щоб не згенерувати помилку? Є два можливих рішення.

Перший заснований на вже відомій нам концепції *символу екранування*, роль якого, як ви пам’ятаєте, виконує символ *зворотного слеша*. Зворотний слеш також може екранувати лапки. Лапки, яким передує зворотний слеш, змінюють своє значення – це вже не роздільник, а просто лапки. Це працюватиме так, як задумано:

```
print("Мені подобається \"ЛНТУ\"")
```

Примітка: всередині рядка є дві екрановані лапки.

Друге рішення може бути трохи дивним. Python може використовувати *апостроф замість лапки*. Будь-який з цих символів може розділяти рядки, але ви повинні бути *послідовні*.

Якщо ви відкриваєте рядок лапкою, то і закривати його потрібно лапкою.

Якщо ви починаєте рядок з апострофа, то ви повинні і закінчувати його апострофом.

Цей приклад також буде працювати:

```
print('Мені подобається "ЛНТУ"')
```

Примітка: вам не потрібно робити жодних екранувань.

Ви вже маєте знати відповідь, точніше, дві можливі відповіді на питання: як вставити апостроф у рядок, розміщений між апострофами? Подумайте.

Нагадаємо, що рядок може бути порожнім – не містити жодного символу.

Порожній рядок все одно залишається рядком: `''` або `""`.

1.6.4 Булеві значення

Щоразу, коли ви запитуєте Python, чи є одне число більшим за інше, це питання призводить до створення певних даних – булевих значень.

Назва походить від імені Джорджа Буля (1815-1864), автора фундаментальної праці «Закони мислення», в якій викладено тлумачення булевої алгебри – розділу алгебри, який оперує виключно двома різними значеннями: `True` і `False`, що позначаються як `1` і `0` відповідно.

Програміст пише програму, а програма задає запитання. Python виконує програму і дає відповіді. Програма повинна вміти реагувати відповідно до отриманих відповідей.

Комп'ютери знають лише два типи відповідей:

– так, це істина (`True`).

– ні, це хибність (`False`).

При використанні булевої логіки неможливо отримати відповіді типу: «Я не знаю» або «Мабуть так, але я точно не знаю».

Ці два Булевих значення мають чіткі позначення в мові Python. Ви не можете нічого змінити – ви повинні прийняти ці позначення такими, як вони є (True та False), включаючи чутливість до регістру.

Задача: Що виведе наступний фрагмент коду?

```
print(True > False)
print(True < False)
```

Нагадаємо, що `True` сприймається як істина, тобто `1`, а `False` – хибність, тобто `0`. Результатом порівняння теж є булеве значення, тобто `True` або `False` (рисунок 1.16).

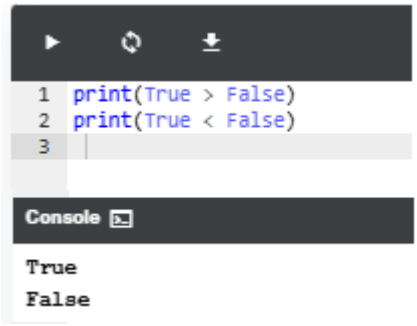


Рисунок 1.16 – Приклад порівняння булевих величин

1.7 Математичні оператори та вирази

Оператор – це символ мови програмування, який здатний оперувати значеннями.

Наприклад, як і в арифметиці, знак + (плюс) – це оператор додавання, який здатний *додавати* два числа.

Оператори, які асоціюються з найбільш відомими арифметичними операціями, наведено в таблиці 1.2.

Таблиця 1.2 – Основні арифметичні оператори

Оператор	**Значення**
+	додавання
-	віднмання
*	множення
/	ділення
//	цілочислене ділення
%	Залишок від ділення
**	піднесення до степеня

Порядок їх розміщення не випадковий. Ми поговоримо про це більш детально, коли їх всіх розглянемо.

Дані та оператори, з’єднані разом, утворюють *вирази*.

Розглянемо приклади (рисунки 1.17-1.18).

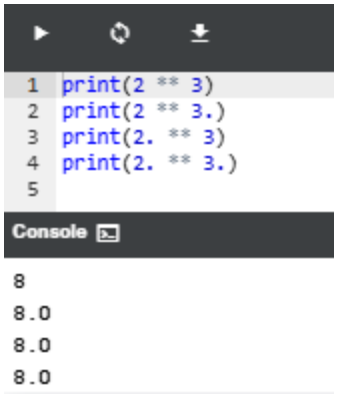


Рисунок 1.17 – Приклад використання оператора піднесення до степеня

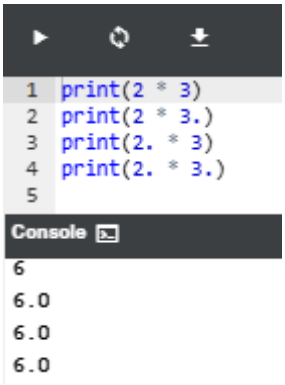


Рисунок 1.18 – Приклад використання оператора множення

Тут з обох сторін від операторів ** або * ми поставили пробіли, але це не є обов'язковим (косметичний ефект, який впливає лише на краще сприйняття коду людиною).

На основі рисунків 1.17 та 1.18 можемо зробити висновки, що при множенні та при піднесенні до степеня:

– коли *обидва* аргументи є цілими числами, результат також буде цілим числом;

– коли *хоча б один* аргумент *є числом з рухомою крапкою*, результат також буде числом з рухомою крапкою.

Знак / (слеш) – оператор *ділення*. Значення до косої риски – це *ділене*, значення після косої риски – *дільник*. Розглянемо приклад (рисунок 1.19).

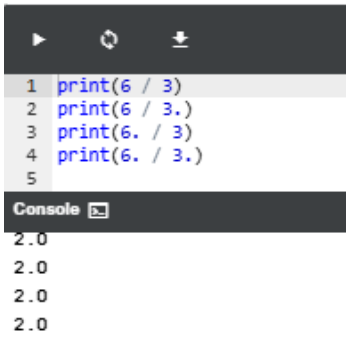


Рисунок 1.19 – Приклад використання оператора ділення

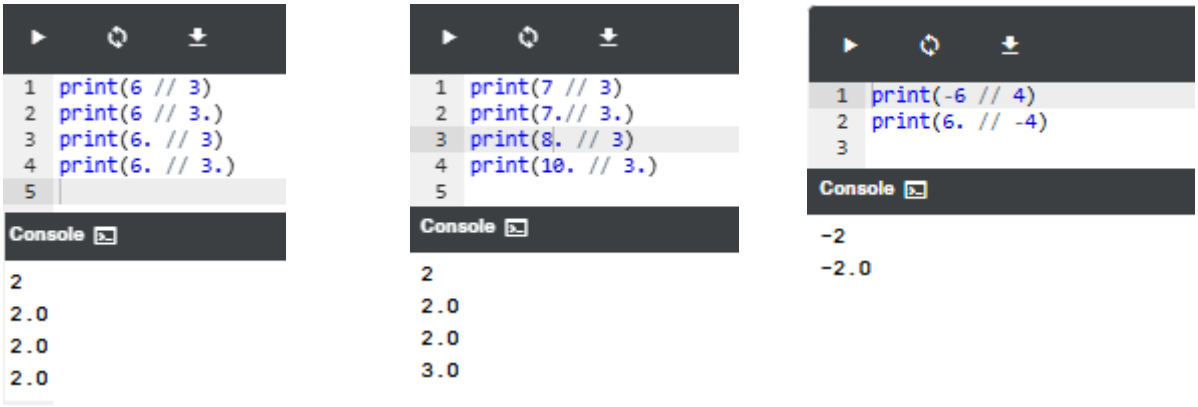
Як бачимо з рисунка 1.19, результат, отриманий в результаті виконання *оператора ділення, завжди є числом з рухомою крапкою*, незалежно від того, чи результат на перший погляд здається числом з рухомою крапкою: 1 / 2, чи він виглядає як чисте ціле число: 2 / 1.

Знак // (подвійна коса риска) є оператором *цілочисельного ділення*. Він відрізняється від стандартного / оператора двома деталями:

– в його результаті відсутня дробова частина – вона відсутня (для цілих чисел) або завжди дорівнює нулю (для чисел з рухомою крапкою); це означає, що результати завжди округляються;

– дотримується правил цілих та чисел з рухомою крапкою.

Розглянемо приклад (рисунок 1.20).



а) б) в)

Рисунок 1.20 – Приклади використання оператора цілочисельного ділення

Як бачите, ділення цілого числа на ціле дає *цілий результат*. У всіх інших випадках результат з рухомою крапкою.

Результат цілочисельного ділення завжди округляється до найближчого цілого числа, яке є меншим за дійсний (не округлений) результат (див. рисунок 1.20, б).

Це дуже важливо: *округлення завжди здійснюється до меншого цілого*.

Примітка: деякі значення є від'ємними (див. рисунок 1.20, в). Результат – дві від'ємні двійки. Реальний (не округлений) результат дорівнює -1.5 в обох

випадках. Однак результати підлягають округленню. Округлення відбувається в бік меншого цілого значення, а меншим цілим значенням є -2, отже: -2 і -2.0.

Цілочисельне ділення також називають *часткою від ділення*.

Результатом операції % є *остача, що залишилась після цілочисельного ділення.*

Іншими словами, це значення, що залишилося після ділення одного числа на інше з виділенням цілої частки. Розглянемо приклад (рисунок 1.21).

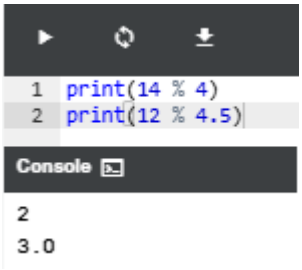


Рисунок 1.21 – Приклади використання оператора остачі від ділення

Як бачимо, для першого виразу результат – двійка. Ось чому:

14 // 4 дає 3 → це ціла частка;

3 * 4 дає 12 → в результаті перемноження частки і дільника;

14 - 12 дає 2 → це остача.

Другий вираз приклад дещо складніший для аналізу, адже 3.0 – не 3, а 3.0. Проте правило працює:

12 // 4.5 дорівнює 2.0,

2.0 * 4.5 дорівнює 9.0,

12 - 9.0 дорівнює 3.0.

З математики відомо, що ділити на нуль не можна. Тому не намагайтеся:

–виконувати ділення на нуль;

–виконувати цілочисельне ділення на нуль;

–знайти остачу від ділення на нуль.

При відніманні оператор мінус очікує два аргументи: лівий (зменшуване в математичному розумінні) і правий (від'ємник в арифметичному розумінні).

З цієї причини оператор віднімання вважається одним з *бінарних* операторів, так само як і оператори додавання, множення та ділення.

Але оператор мінус можна використовувати і по-іншому (*унарно*) – подивіться на приклад, наведений на рисунку 1.22.

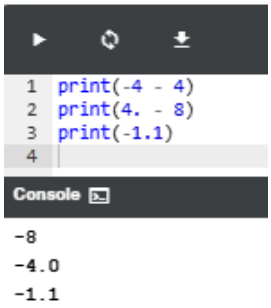


Рисунок 1.22 – Приклади використання бінарних та унарних операторів

Як видно з рисунку 1.22, перші два рядки містять бінарні оператори, а третій – унарний.

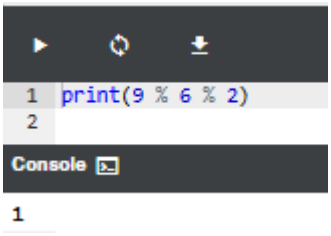
Дуже часто в одному виразі можна зустріти більше одного оператора.

Python чітко визначає пріоритети всіх операторів і вважає, що оператори з вищим пріоритетом виконують свої операції раніше ніж оператори з нижчим пріоритетом. В таблиці 1.2. (див. вище) оператори наведені від найнижчого до найвищого пріоритету.

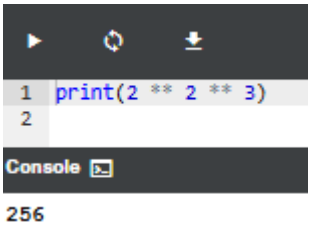
Асоціативність оператора визначає порядок обчислень, які виконуються деякими операторами з однаковим пріоритетом і стоять поруч в одному виразі.

Більшість операторів мови Python мають лівосторонню асоціативність, що означає, що обчислення виразу виконується зліва направо.

Розглянемо приклади (рисунок 1.23).



а)



б)

Рисунок 1.23 – Приклади різної асоціативності операторів

Для виразу з рисунку 1.23, а) існує два варіанта обчислення:

1) зліва направо: спочатку 9 % 6 дають 3, а потім 3 % 2 дають 1;

2) справа наліво: спочатку 6 % 2 дає 0, а потім 9 % 0 викликає фатальну *помилку*.

Так як ми отримали результат 1, то оператор % має *ліву асоціативність*.

Для виразу з рисунку 1.23, б) можливі два результати:

1) 2 ** 2 → 4; 4 ** 3 → 64

2) 2 ** 3 → 8; 2 ** 8 → 256

Так як ми отримали результат 256 (див. рисунок 1.23, б), то в операторі *піднесення до степеня використовується правостороння асоціативність.*

Розгляньте приклад, наведений на рисунку 1.24 і подумайте чому отримали такий результат (подумайте про асоціативність).

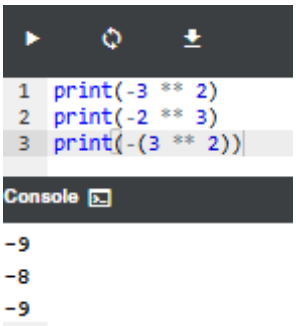


Рисунок 1.24 – Приклади використання оператора піднесення до степеня та унарного мінуса

Завжди можна використовувати *круглі дужки*, які можуть змінити звичайний порядок обчислень.

Відповідно до правил арифметики, спочатку завжди обчислюються підвирази в дужках.

Ви можете застосовувати стільки круглих дужок, скільки вам потрібно, їх часто використовують для покращення читабельності виразу, навіть якщо вони не змінюють порядок виконання операцій.

1.8 Змінні

Мови програмування також дозволяють визначати змінні. *Змінна* – це не що інше, як зарезервоване місце пам’яті для зберігання значень. Це означає, що при створенні змінної виділяється деякий простір (рисунок 1.25).

Рисунок 1.25 – Ілюстрація аналогії виділення місця в пам’яті для змінної

Сама назва («змінні») підказує, що вміст цих контейнерів («коробок») можна змінювати у (майже) будь-який спосіб.

Кожна змінна Python має ім’я та значення (вміст).

1.8.1 Імена змінних

При *присвоєнні імені змінній* необхідно дотримуватись деяких чітких правил:

– ім’я змінної має складатися з великих або малих літер, цифр та символу _ (підкреслення);

– ім’я змінної має починатися з літери;

– символ підкреслення – це буква;

– великі та малі літери розглядаються як різні (трохи інакше, ніж у реальному світі – `Alice` та `ALICE` – це одне і теж ім’я, але в Python це два різних імені змінних, а отже, дві різні змінні);

– ім’я змінної не має бути жодним з зарезервованих слів Python (ключових слів – про це ми розповімо пізніше).

Зверніть увагу, що такі ж вимоги поширюються і на імена функцій.

Python не накладає обмежень на довжину імен змінних, але це не означає, що довге ім’я змінної краще за коротке.

Наведемо кілька коректних, але не завжди зручних назв змінних:

```
MyVariable
i
l
t34
Exchange_Rate
counter
days_to_christmas
TheNameIsTooLongAndHardlyReadable
_
```

Ці назви змінних також правильні:

```
Adiós_Señora
sûr_la_mer
Einbahnstraße
змінна.
```

Python дозволяє використовувати не тільки латинські літери, але й символи, характерні для мов, що використовують інші алфавіти.

А тепер про деякі помилкові імена:

– `10t` (не починається з літери)

– `!important` (не починається з літери)

– exchange rate (містить пробіл).

Примітка: PEP 8 -- Style Guide for Python Code рекомендує наступний порядок іменування змінних та функцій у мові Python:

– імена змінних повинні бути написані малими літерами, з розділенням слів символом підкреслення для покращення читабельності (наприклад, var, my_variable)

– імена функцій дотримуються тих самих правил, що й імена змінних (наприклад, `fun, my_function`)

– також можна використовувати змішаний регістр (наприклад, `myVariable`), але тільки в ситуаціях, коли це вже є загальноприйнятим стилем, щоб зберегти зворотну сумісність з прийнятою конвенцією.

1.8.2 Ключові слова

Подивіться на список слів, які відіграють особливу роль у кожній програмі Python.

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Вони називаються ключовими *словами* або (точніше) зарезервованими ключовими словами. Вони зарезервовані, тому що ви не повинні використовувати їх як імена: ні для ваших змінних, ні для функцій, ні для будь-яких інших іменованих об'єктів, які ви захочете створити.

Значення зарезервованого слова є *заздалегідь визначеним* і не може бути змінено жодним чином.

На щастя, через те, що Python чутливий до регістру, ви можете модифікувати будь-яке з цих слів, змінивши регістр будь-якої літери, створивши таким чином нове слово, яке не є зарезервованим.

Наприклад, *ви не можете назвати* свою змінну так:

```
import
```

Але ви можете назвати свою змінну так:

```
Import
```

Ви можете використовувати змінну для зберігання будь-якого значення будь-якого типу даних.

1.8.3 Створення змінних

Змінна утворюється в результаті присвоєння їй значення. На відміну від інших мов, її не потрібно оголошувати якимось особливим чином.

Якщо присвоїти неіснуючій змінній будь-яке значення, змінна буде автоматично створена. Більше нічого не потрібно робити.

Створення (іншими словами, її синтаксис) надзвичайно простий: достатньо використати ім’я потрібної змінної, потім знак рівності (=) і значення, яке ви хочете надати змінній.

Розглянемо приклад (рисунок 1.26).

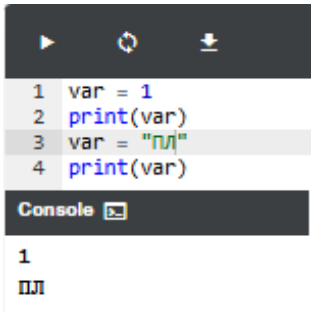


Рисунок 1.26 – Приклад створення та використання змінної

Перший рядок створює змінну з іменем `var` і присвоює літерал з цілим значенням рівним `1`.

Другий виводить на консоль значення новоствореної змінної.

Третій замінює значення змінної (спочатку вона містила ціле значення 1, а тепер рядок «ПЛ»).

Четвертий виводить на консоль поточне значення змінної, тобто після змін.

Отже, значення змінної може змінюватися так часто, як ви цього хочете або потребуєте. Воно може бути цілим числом зараз, або числом з рухомою крапкою вже через мить, в подальшому перетворюючись на рядок.

Знак рівності фактично є *оператором присвоювання*. Хоча це може здатися дивним, оператор має простий синтаксис і однозначне тлумачення.

Він присвоює значення свого правого аргументу лівому, при цьому правий аргумент може бути яким завгодно складним виразом, що включає в себе літерали, оператори і вже визначені змінні.

Спробуймо розв’язати задачу. Знайти гіпотенузу прямокутного трикутника, коли відомі довжини його катетів.

Як відомо з математики, квадрат гіпотенузи дорівнює сумі квадратів катетів.

Отже, довжина гіпотенузи – це корінь квадратний з сумі квадратів катетів.

В мовах програмування коли виникає потреба знайти якийсь корінь, то його замінюють степенем за формулою:

$$\sqrt[n]{x} = x^{\frac{1}{n}} \tag{1.1}$$

Відповідно $\sqrt[2]{x} = x^{\frac{1}{2}}$.

Замість x підставимо суму квадратів катетів і отримаємо формулу:

$$c = \left(a^2 + b^2\right)^{\frac{1}{2}} \tag{1.2}$$

Присвоїмо довільні значення катетів відповідним змінним, виконаємо обчислення і отримаємо результат (рисунок 1.27).

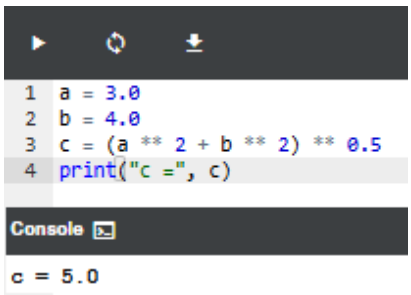


Рисунок 1.27 – Приклад використання змінних для розв'язування задач

У першому рядку *створюється нова змінна* з іменем `a` і їй присвоюється значення 3.0.

У другому рядку *створюється нова змінна* з іменем `b` і їй присвоюється значення 4.0.

В третьому рядку *створюється нова змінна* з іменем `c` і їй присвоюється значення, обчислене за формулою (1.2).

В четвертому рядку виводиться значення змінної с за допомогою функції `print()`.

1.9 Розширені оператори присвоєння

Якщо нам потрібно x помножити на 2, ми можемо використати такий код:

```
x = x * 2
```

Python пропонує нам скорочений спосіб запису цієї операції наступним чином:

```
x *= 2
```

В загальному випадку якщо `op` є оператором з двома аргументами (це дуже важлива умова), і оператор використовується в наступному контексті...:

```
variable = variable op expression
```

... тоді його можна спростити і представити наступним чином:

```
variable op= expression
```

Можна поєднувати арифметичні оператори з привласненням, розміщуючи оператор перед знаком =.

Вираз `a -= 3` аналогічний виразу `a = a - 3`.

Замість знаку – можуть стояти інші оператори: `+, *, /, //, %/` Розглянемо приклади (таблиця 1.3).

Таблиця 1.3 – Приклади використання розширених операторів присвоєння

Звичайний вираз	***Розширені оператори присвоєння***
`i=i+2*j`	`i+=2*j`
`var=var/2`	`var/=2`
`rem=rem%10`	`ren%=10`
`j=j-(i+var+rem)`	`j-=(i+var+rem)`
`x=x**2`	`x**=2`

1.10 Коментарі в Python

По мірі збільшення розмірів програм рано чи пізно код стане складніше читати. Для підвищення зрозумілості коду, його корисно доповнювати коментарями на природній мові, і більшість мов програмування, у тому числі Python, надають таку можливість.

Коментування коду вважається правилом «хорошого тону».

Коли над програмою працює один програміст, то відсутність коментарів компенсується хорошим знанням коду, але при роботі в команді, за рідкісними винятками, коментарі просто необхідні.

Коментар – це фрагмент тексту у програмі, який буде проігнорований інтерпретатором Python. Можна використовувати коментарі, щоб дати пояснення до коду, зробити якісь позначки для себе, або для чогось ще.

Коментар позначається символом #;

все, що знаходиться після # до кінця поточного рядка, є коментарем. Зазвичай коментар розташовується на окремому рядку:

```
# Підрахунок процентного співвідношення двох величин: 20 та 80
print (100 * 20 / 80, "%")
```

Отримаємо:

```
25.0 %
```

Або на тому самому рядку, що і код, який потрібно пояснити:

```
print (100 * 20 / 80, "%") # Підрахунок процентного
          # співвідношення двох величин: 20 та 80
```

Отримаємо:

```
25.0 %
```

Якщо вам потрібен коментар, який охоплює кілька рядків, ви повинні поставити хештег перед усіма рядками:

```
# Ця програма обчислює гіпотенузу c.
# a і b – довжини катетів.
a = 3.0
b = 4.0
c = (a ** 2 + b ** 2) ** 0.5  # Ми використовуємо **
                              # замість квадратного кореня.
print("c =", c)
```

Символ # має багато імен: хеш, шарп, фунт або октоторп.

Коментар діє тільки до кінця рядка, на якому він розташовується.

Однак якщо символ # знаходиться всередині текстового рядка, він стає простим символом #:

```
print("Без коментарів #")
```

Отримаємо:

Без коментарів #

Коментарі можуть бути корисними ще в одному випадку – ви можете використовувати їх для *позначення частини коду*, яка в даний момент з якихось причин не потрібна. Подивіться на приклад нижче, якщо ви *розкоментуєте* виділений рядок, це вплине на результати роботи коду:

```
# Це тестова програма.
x = 1
y = 2
# y = y + x
print(x + y)
```

Це часто робиться під час тестування програми, щоб ізолювати місце, де може бути прихована помилка.

Порада. Якщо ви хочете швидко закоментувати або розкоментувати кілька рядків коду, виберіть рядок (рядки), які ви хочете закоментувати, і скористайтеся наступною комбінацією клавіш: CTRL + / .

Коли це можливо і доцільно, слід давати само-коментовані імена змінним, наприклад, якщо у вас є дві змінні: одна для збереження довжини, інша для збереження ширини, то імена змінних `length` і `width` будуть більш доречними, ніж `myvar1` і `myvar2`.

Важливо використовувати коментарі, для того щоб зробити програми більш зрозумілими; а також використовувати в коді читабельні та змістовні імена змінних. Однак не менш важливо: не використовувати назви змінних, які можуть заплутати; і не залишати коментарі, які містять невірну або некоректну інформацію.

Коментарі можуть бути корисними, коли ви читаєте власний код через деякий час (можете нам повірити, розробники забувають, що робить їх власний код), або коли інші читають ваш код (вони можуть допомогти їм швидше зрозуміти, що роблять ваші програми і як вони це роблять).

1.11 Функція input()

Функція `input()` здатна зчитувати дані, що вводяться користувачем, та передавати ці дані у програму, що виконується.

Програма може маніпулювати даними, роблячи код по-справжньому інтерактивним.

Практично всі програми *зчитують і обробляють дані*. Програма, яка не отримує вхідних даних від користувача, називається *глухою програмою*.

Розглянемо приклад (рисунок 1.28).

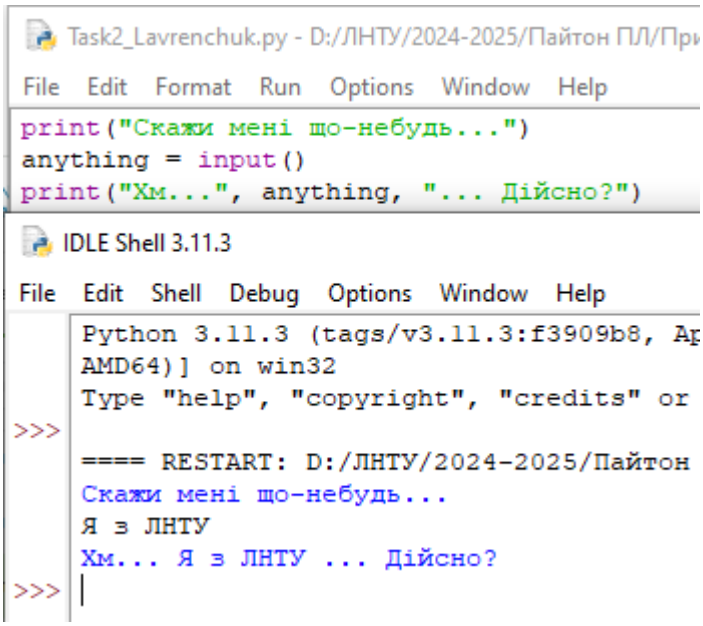


Рисунок 1.28 – Приклад використання функції `input()`

Тут продемонстровано дуже простий варіант використання функції `input()`:

– програма пропонує користувачу ввести деякі дані з консолі (найімовірніше за допомогою клавіатури, хоча можливе також введення даних за допомогою голосу або зображень);

–функція `input()` викликається без аргументів (це найпростіший спосіб використання функції); функція переведе консоль в режим введення; ви побачите миготливий курсор і зможете здійснити натискання клавіш,

завершивши введення натисканням клавіші `Enter`; всі введені дані будуть передані у вашу програму за результатом виконання функції;

– результат потрібно присвоїти змінній; це важливо – пропуск цього кроку призведе до втрати введених даних;

– потім за допомогою функції `print()` виводимо отримані дані з деякими додатковими коментарями.

Функція `input()` може видавати запит користувачу без допомоги функції `print()`. Розглянемо приклад (рисунок 1.29).

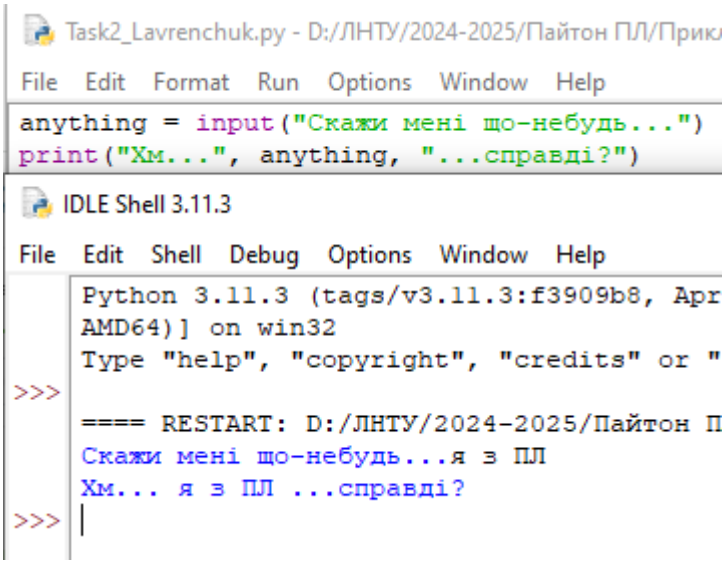


Рисунок 1.29 – Приклад використання функції `input()` з аргументом

– функція `input()` викликається з одним аргументом – це рядок, що містить повідомлення;

– повідомлення буде виведено на консоль до того, як користувачу буде надана можливість ввести будь-що;

– після цього функція `input()` виконає свою задачу (візьме введений користувачем текст та покладе його в змінну `anything .`

Такий варіант виклику `input()` спрощує код та робить його більш зрозумілим.

Результатом функції `input()` є *рядок*. Рядок містить всі символи, які користувач ввів з клавіатури. Це не ціле число і не число з рухомою крапкою. Це означає, що *ви не можете використовувати його як аргумент будь-якої*

арифметичної операції, наприклад, ви не можете використовувати ці дані, щоб звести їх у квадрат, поділити на будь-що або поділити на них що-небудь.

```
anything = input("Введіть число: ")
something = anything ** 2.0 # так не можна!
print(anything, "в степені 2 дорівнює", something)
```

Ми намагались застосувати оператор ** до `'str'` (рядка) у комбінації з `'float'`. Це заборонено.

1.12 Приведення (перетворення) типів даних

Python пропонує дві прості функції для визначення типу даних і вирішення цієї проблеми –`int()` і `float()`.

Їхні назви говорять самі за себе:

– функція `int()` отримує один аргумент (наприклад, рядок: `int(string)`) і намагається перетворити його в ціле число; якщо їй це не вдається, то вся програма завершиться помилкою;

– функція `float()` отримує один аргумент (наприклад, рядок: `float(string)`) і намагається перетворити його у число з рухомою крапкою.

Це дуже просто і дуже ефективно. Крім того, можна викликати будь-яку з функцій, передавши їм безпосередньо результати `input()`. Немає необхідності використовувати будь-яку змінну як проміжне сховище.

Розглянемо приклад (рисунок 1.30).

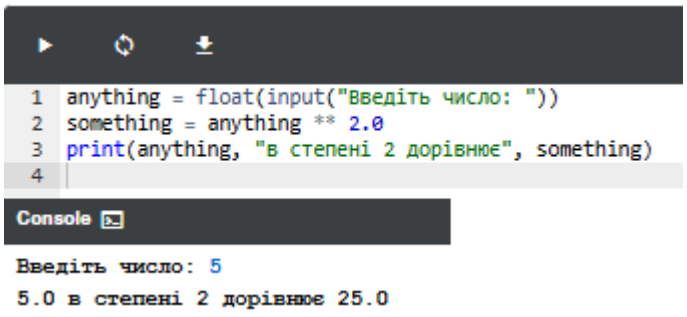


Рисунок 1.30 – Приклад використання функції `input()` та приведення типу

Бачите, як введений користувачем рядок потрапляє з `input()` в `print()`?

Функціонал, що складається з тріо `input()-int()-float()`, відкриває нам багато нових можливостей.

Повернемося до прикладу, наведеного на рисунку 1.27 (задача про довжину гіпотенузи) та змінимо його так, щоб можна було задавати довільні довжини катетів з клавіатури (рисунок 1.31).

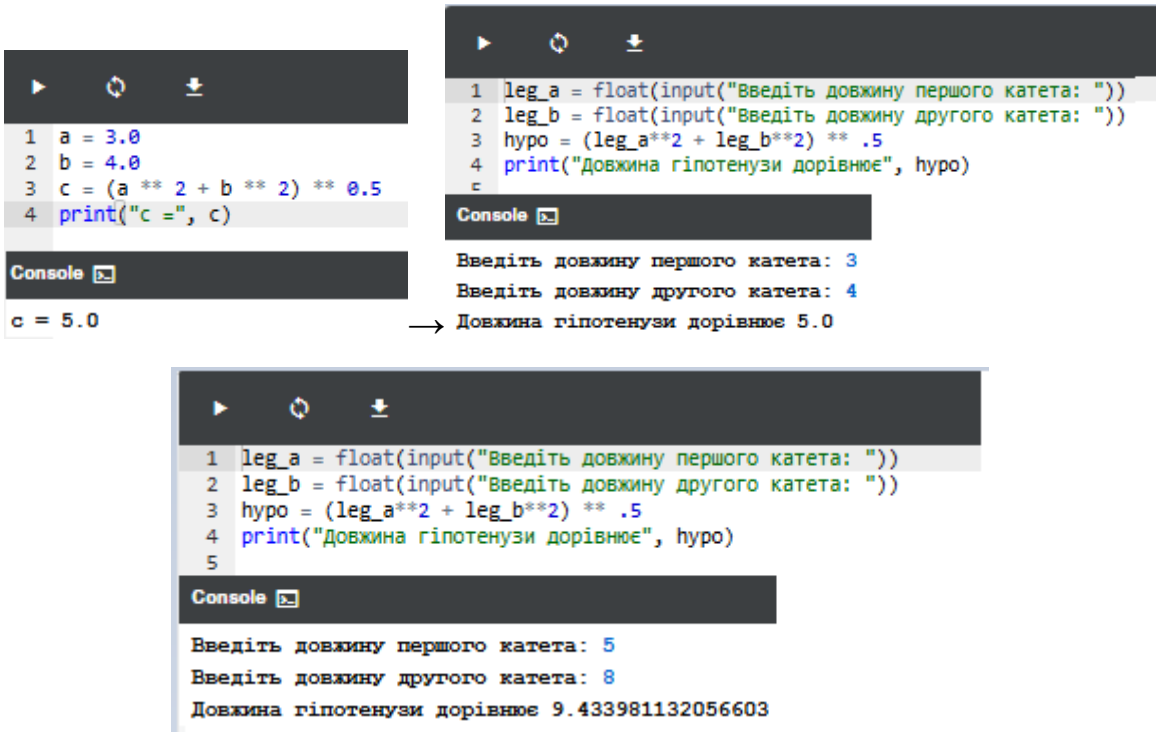


Рисунок 1.31 – Приклад використання функції `input()` для введення різних даних без зміни коду

Програма запитує у користувача довжини обох катетів, обчислює гіпотенузу і виводить результат.

Зверніть увагу на програму в редакторі, змінна `hypo` використовується тільки з однією метою – для збереження обчисленого значення до виконання наступного рядка коду.

Оскільки функція print() отримує в якості аргументу вираз, то змінну можна видалити з коду (рисунок 1.32).

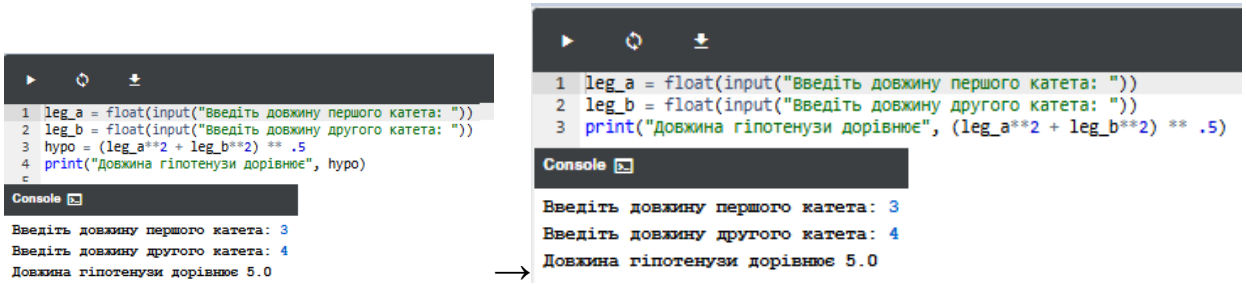


Рисунок 1.32 – Приклад скорочення коду програми

1.13 Конкатенація та реплікація стрічок

Знак + (плюс) при застосуванні до двох рядків стає *оператором склеювання* (конкатенації):

```
string + string
```

Він просто конкатенує (з'єднує) два рядки в один. Звичайно, як і його арифметичний побратим, він може неодноразово використовуватися в одному виразі, і в такому випадку поводиться за правилом *лівої асоціативності*.

На відміну від свого арифметичного побратима, оператор конкатенації *не є комутативним*, тобто "ab" + "ba" – це не те ж саме, що "ba" + "ab".

Не забувайте: якщо ви хочете, щоб знак + був *конкатенатором*, а не суматором, ви повинні переконатися, що *обидва його аргументи є рядками*. Тут не можна змішувати типи.

Ця проста програма демонструє другу функцію використання знаку +:

Розглянемо приклад (рисунок 1.33).

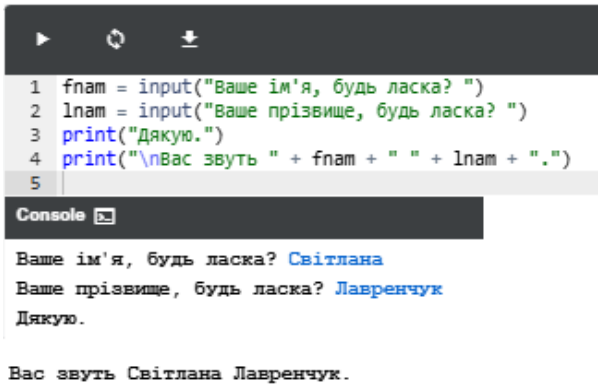


Рисунок 1.33 – Приклад конкатенації двох слів

Примітка: використання + для склеювання рядків дозволяє побудувати виведення більш точно, ніж за допомогою звичайної функції `print()`, навіть якщо вона доповнена аргументами з ключовими словами `end=` і `sep=`.

Знак * (зірочка) при застосуванні до рядка і числа (або числа і рядка, оскільки їх можна міняти місцями) стає оператором *реплікації* :

```
string * number
number * string
```

Він повторює рядок стільки ж разів, скільки задається числом. Наприклад:

```
"James" * 3  #дає "JamesJamesJames"
3 * "an" #дає "ananan"
5 * "2" (or "2" * 5) #дає "22222" (не 10!)
```

Якщо використано в такій ролі число, менше або рівне нулю, то виводеться *порожній рядок*.

За допомогою використання конкатенації, реплікації можна «намалювати» примітивний прямокутник в консолі (рисунок 1.34).

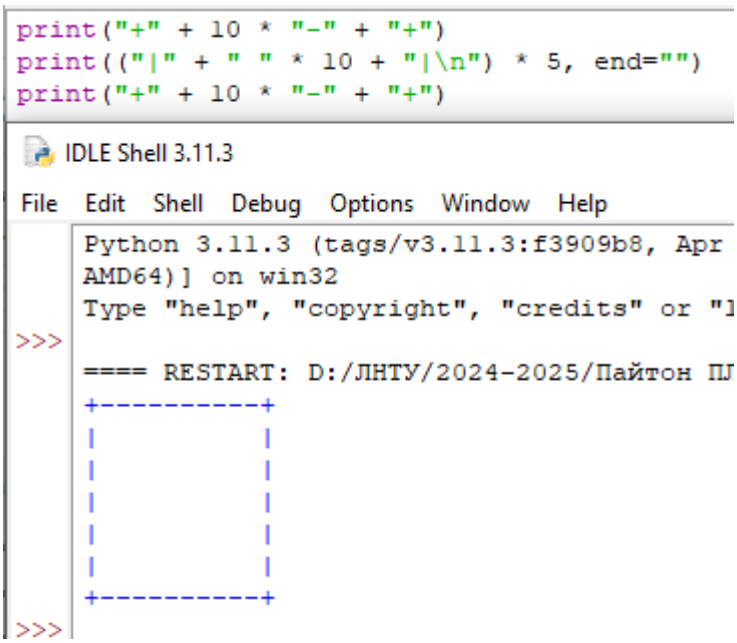


Рисунок 1.34 – Приклад використання конкатенації та реплікації

Також можна *перетворити число в рядок*. Функція, яка вміє це робити, називається `str()`:

`str(number)`

Можемо ще змінити розв'язок задачі про гіпотенузу (рисунок 1.35).

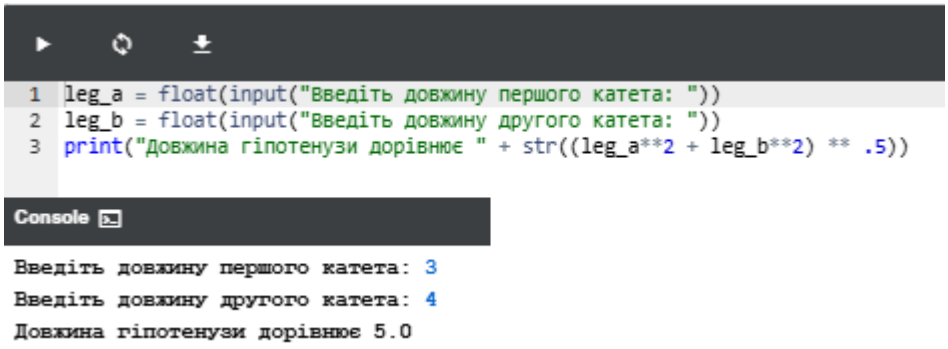


Рисунок 1.35 – Приклад використання функції `str()`

Завдяки функції `str()` ми можемо *передати у функцію* `print()` *весь результат одним рядком*, відмовившись від ком.

Контрольні питання

1. Що таке мова програмування Python і які основні етапи її розвитку?
2. Назвіть основні переваги та недоліки мови Python.
3. Які завдання найчастіше вирішуються за допомогою Python? Наведіть приклади застосувань.
4. Для чого використовується функція `print()` у Python? Який її синтаксис і які аргументи вона може приймати?
5. Що таке екрануючий символ у Python? Наведіть приклади.
6. Назвіть основні типи даних у Python. Які значення належать до кожного з них?
7. Яка різниця між цілими (int) і дійсними (float) числами в Python?
8. Як у Python представляються логічні (булеві) значення? Яке їх призначення?
9. Які математичні оператори підтримує Python? Наведіть приклади виразів із їх використанням.
10. Що таке змінна у Python? Як створити змінну та надати їй значення?
11. Які правила іменування змінних у Python? Чому не можна використовувати ключові слова як імена змінних?
12. У чому полягає відмінність між звичайним оператором присвоєння (=) та розширеними операторами (+=, -=, *= тощо)?
13. Яке призначення коментарів у Python? Як вони створюються?
14. Для чого використовується функція input()? Як з її допомогою вводити числові та текстові дані?
15. Що таке приведення типів даних у Python? Як здійснити перетворення типів, наприклад, зі str у int або з float у str?

ТЕМА 2. Оператори порівняння та розгалужені алгоритми

2.1 Оператори порівняння

2.1.1 Оператор рівності

Щоб порівняти дві величини на рівність використовується оператор `==` (дорівнює дорівнює).

Є важлива відмінність:

`=` це *оператор присвоєння*, наприклад, `a = b` присвоює `a` значення `b`;

`==` запитує, *чи рівні* ці величини? Тобто `a == b` порівнює `a` та `b`.

Це бінарний оператор *з лівою асоціативністю*. Він потребує двох аргументів і перевіряє їх рівність.

В таблиці 2.1 наведено приклади використання оператора рівності.

Таблиця 2.1 – Приклади порівнянь на рівність

Порівняння	***Результат порівняння***	***Пояснення***
`2 == 2`	`True`	2 дорівнює 2
`2 == 2.`	`True`	Python може перетворити ціле число в його дійсний еквівалент і тоді порівнює, тому такий результат
`1 == 2`	`False`	1 не дорівнює 2

Оператор `==` (дорівнює) порівнює значення двох операндів. Якщо вони рівні, то результат порівняння дорівнює `True`. Якщо вони не рівні, то результат порівняння дорівнює `False.`

Розглянемо приклад, наведений на рисунку 2.1.

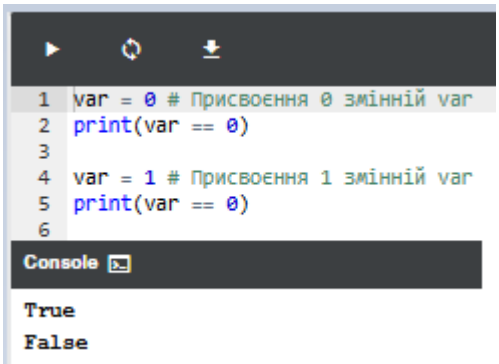


Рисунок 2.1 – Приклад використання оператора порівняння

Якщо змінна багато разів змінюється під час виконання програми, то результат порівняння теж змінюється.

Вираз

```
black_sheep == 2 * white_sheep
```

потрібно сприймати так:

```
black_sheep == (2 * white_sheep)
```

Це пов’язано з низьким пріоритетом оператора `==`.

2.1.2 Оператор не дорівнює (!=)

Оператор `!=` (не дорівнює) теж порівнює значення двох операндів. Різниця ось у чому: якщо вони рівні, результатом порівняння є `False`. Якщо вони не рівні, результатом порівняння буде `True`.

Приклад роботи з оператором `!=` наведений на рисунку 2.2.

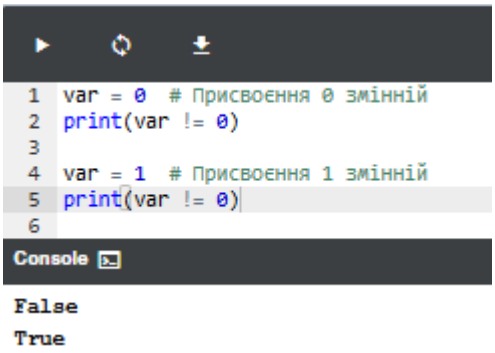


Рисунок 2.2 – Приклад використання оператора !=

Як бачимо, оператори == та != дають протилежні результати.

2.1.3. Інші оператори порівняння

Ви також можете зробити порівняння, використовуючи оператор `>` (більше ніж). Якщо ви хочете дізнатися, чи більше чорних овець, ніж білих, то можна записати це так:

```
black_sheep > white_sheep # Більш ніж
```

`True` це підтверджує; `False` заперечує.

Оператор більше ніж має інший спеціальний, *нестрогий варіант*, ніж в класичній арифметичній нотації, і позначається він інакше: `>=` (більше або дорівнює).

Тут два послідовні знаки, а не один.

Обидва ці оператори (строгий і нестрогий), як і два інших, що розглядаються в наступному розділі, є бінарними операторами з лівою асоціативністю, і їх пріоритет вищий, ніж у операторів `==` та `!=`.

Якщо ми хочемо дізнатися, чи потрібно нам одягти теплу шапку, ми задаємося наступним питанням:

```
centigrade_outside >= 0.0  # Більше або дорівнює
```

Використовуються також оператори: < (менше) і його менш строгий побратим: <= (менше або дорівнює).

Подивіться на цей простий приклад:

```
current_velocity_mph < 85  # Менше ніж
current_velocity_mph <= 85  # Менше або дорівнює
```

2.1.4 Приклади використання операторів порівняння

Результат порівняння можна запам’ятати (зберегти у змінній) і використати пізніше:

```
answer = number_of_lions >= number_of_lionesses
```

Можна також використати отриману відповідь для прийняття рішення про подальшу роботу програми.

Оператори порівняння мають низький пріоритет (таблиця 2.2).

Таблиця 2.2 – Таблиця пріоритетів операторів

Пріоритет	***Оператор***	***Примітка***
1	+, -	унарні
2	**	
6	*, /, //, %	
4	+, -	бінарні
5	<, <=, >, >=	
6	= =, !=	

На рисунку 2.3 наведено приклад програми, яка приймає параметр `n` у якості вхідних даних, яке є цілим числом, і друкує `False`, якщо `n` менше за `100` та `True`, якщо n більше або дорівнює `100`.

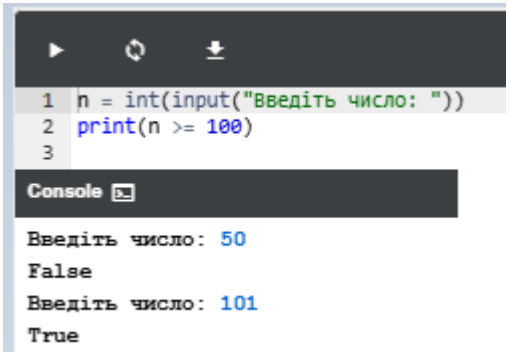


Рисунок 2.3 – Приклад використання оператора >=

2.2 Умови та умовне виконання

2.2.1 Умовний оператор if

Програми повинні вміти виконувати різні дії, залежно від введених даних. Для прийняття рішення програма перевіряє, істинна чи хибна певна умова.

Перевірка умов та прийняття рішень за результатами цієї перевірки називається розгалуженням (branching).

Програма в такий спосіб вибирає, якою з можливих гілок їй рухатися далі. У Python перевірка умови здійснюється за допомогою ключового слова `if`. Розглянемо програму (рисунок 2.4).

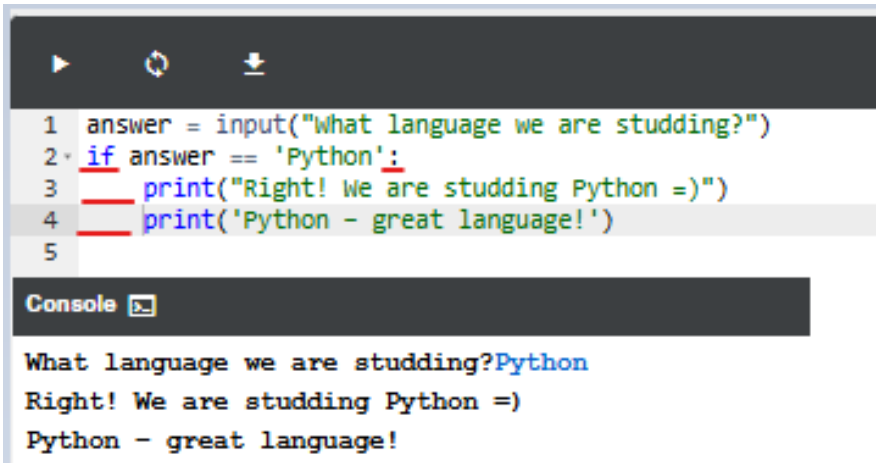


Рисунок 2.4 – Приклад використання ключового слова if

Двокрапка (`:`) в кінці рядка з інструкцією `if` повідомляє інтерпретатору Python, що далі знаходиться блок команд. До блоку команд входять усі рядки з відступом під рядком з інструкцією `if`, аж до наступного рядка без відступу.

Якщо умова є істинною, виконується весь розташований нижче блок. У попередньому прикладі блок інструкцій складає третій і четвертий рядки програми. Відступ робимо за допомогою кнопки `Tab` на клавіатурі.

Блоком коду називають об'єднані один з одним рядки. Вони завжди пов'язані з певною частиною програми (наприклад, з інструкцією `if`). У Python блоки коду формуються за допомогою відступів (рисунок 2.5).

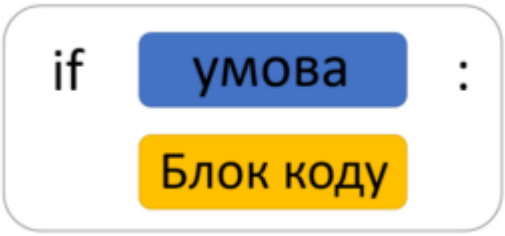


Рисунок 2.5 – Блок коду

Логіку роботи умовного оператора можна описати наступним чином:

```
if true_or_not:
    do_this_if_true
```

Цей умовний оператор складається з наступних, строго необхідних, елементів в такому і тільки в такому порядку:

– ключове слово `if`;

– один або кілька пробілів;

– вираз (питання або відповідь), значення якого буде інтерпретуватися виключно в термінах `True` (коли його значення відмінне від нуля) і `False` (коли воно дорівнює нулю);

– двокрапка, за якою слідує новий рядок;

– інструкція або набір інструкцій з відступом (принаймні одна інструкція є обов'язковою); відступ може бути досягнутий двома способами – вставкою певної кількості пробілів (рекомендується використовувати чотири пробіли відступу), або за допомогою символу табуляції (`Tab`); примітка: *якщо в блоці більше однієї інструкції, то відступ повинен бути однаковим у всіх рядках*; хоча це може виглядати однаково, якщо ви використовуєте табуляцію упереміш з

пробілами, важливо зробити всі відступи абсолютно однаковими – Python 3 *не допускає змішування пробілів і табуляції* для створення відступів.

Логіку роботи умовного оператора:

– якщо вираз true_or_not є істинним (тобто його значення не дорівнює нулю), то буде виконано оператор(и) з відступом;

– якщо вираз true_or_not не є істинним (тобто його значення дорівнює нулю), то оператор(и) з відступом буде опущено (проігноровано), а наступною інструкцією, що виконуватиметься, буде та, що знаходиться вище за початковий рівень відступу.

Приклад використання ключового слова `if` разом з операторами порівняння наведено на рисунку 2.6.

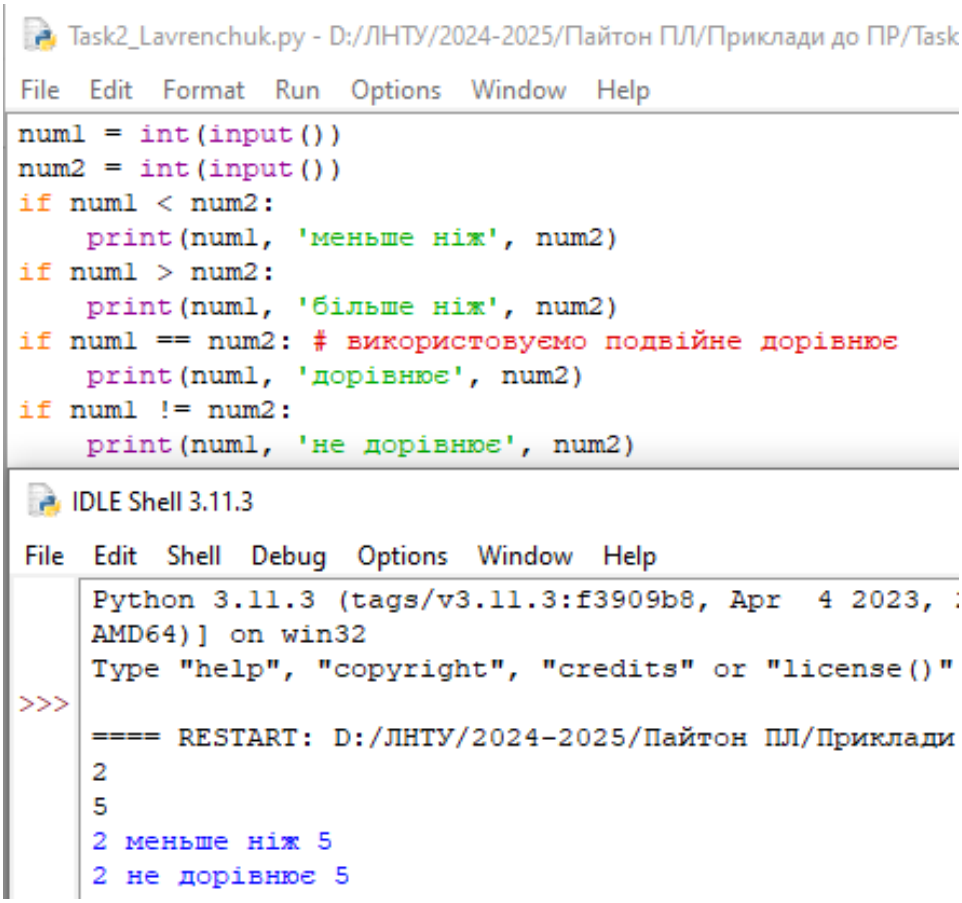


Рисунок 2.6 – Приклад використання if разом з операторами порівняння

2.2.2 Умовний оператор if-else

Програма з рисунку 2.4 виводить текст у випадку, якщо умова є істинною. Але якщо умова хибна, то програма нічого не виводить. Для того, щоб

забезпечити можливість виконувати будь-що, якщо умова виявилася хибною, ми використовуємо ключове слово `else`. Розглянемо приклад (рисунок 2.7).

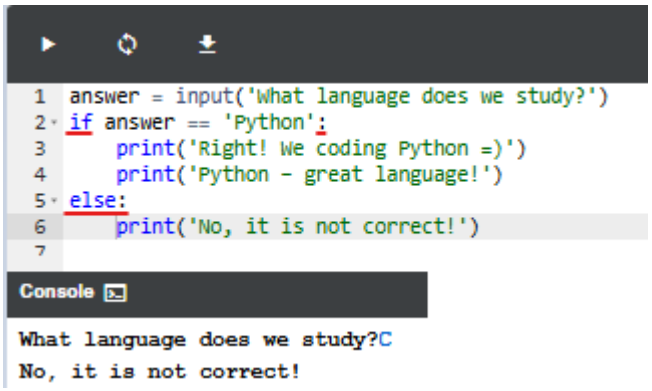


Рисунок 2.7 – Приклад використання ключових слів `if-else`

У цій програмі ми обробляємо відразу два випадки: якщо умова істинна (користувач ввів «Python»), і якщо умова хибна (користувач ввів що завгодно, крім «Python»). Загальна структура цієї програми наведена на рисунку 2.8.

Рисунок 2.8 – Схема використання ключових слів `if-else`

Саме відступ повідомляє інтерпретатору Python, де починається і закінчується блок коду.

Таким чином, з'являється нове слово: `else` – це ключове слово.

Частина коду, яка починається з `else`, говорить про те, що робити, якщо умова, задана для `if`, не виконується (зверніть увагу на двокрапку після цього слова).

Робота оператора `if-else` виглядає наступним чином:

– якщо умова приймає значення True (її значення не дорівнює нулю), то виконується блок після першої двокрапки, і на цьому виконання умовного оператора завершується;

– якщо умова набуває значення False (дорівнює нулю), то виконується після `else`, і на цьому виконання умовного оператора завершується.

2.2.3 Ланки порівняння

Оператори порівняння в Python можна об'єднувати в ланки (на відміну від більшості інших мов програмування, де для цього потрібно використовувати логічні зв'язки), наприклад, `a == b == c` або `1 <= x <= 10`.

Наступний код перевіряє, чи знаходиться значення змінної `age` в діапазоні від `3` до `6`:

```
age = int(input())
if 3 <= age <= 6:
print('You are baby')
```

Код, що перевіряє рівність трьох змінних, може мати такий вигляд:

```
if a == b == c:
print('digits is equal')
else:
print('digits is not equal')
```

2.2.4 Транзитивність

Операція рівності є транзитивною. Це означає, що якщо `a == b` та `b == c`, то з цього випливає, що `a == c`. Саме тому попередній код, що перевіряє рівність трьох змінних, працює як належить.

З курсу математики вам можуть бути знайомі інші приклади транзитивних операцій:

– відношення порядку: якщо `a>b` та `b> c`, то `a> c`;

– паралельність прямих: якщо `a ‖ b` та `b ‖ c`, то `a ‖ c`;

– подільність: якщо `a` ділиться на `b` та `b` ділиться на `c`, `a` ділиться на `c`.

Наочно транзитивність відношення порядку можна зрозуміти на такому прикладі: якщо сусід зліва старший за вас (`a > b`), а ви старший за сусіда справа (`b > c`), то сусід зліва точно старший за сусіда справа (`a > c`).

Операція нерівності (`!=`), на відміну операції рівності (==), є нетранзитивною.

Тобто з того, що `a! = b` і `b! = c`, зовсім не випливає, що `a! = c`. Справді, якщо вас звуть не так, як сусіда зліва і не так, як сусіда справа, то немає гарантії, що в обох сусідів не будуть однакові імена.

Таким чином, наступний код зовсім не перевіряє той факт, що всі три змінні різні (рисунок 2.9):

```
if a != b != c:
    print(‘digits is equal’)
else:
    print(‘digits is not equal ‘)
```

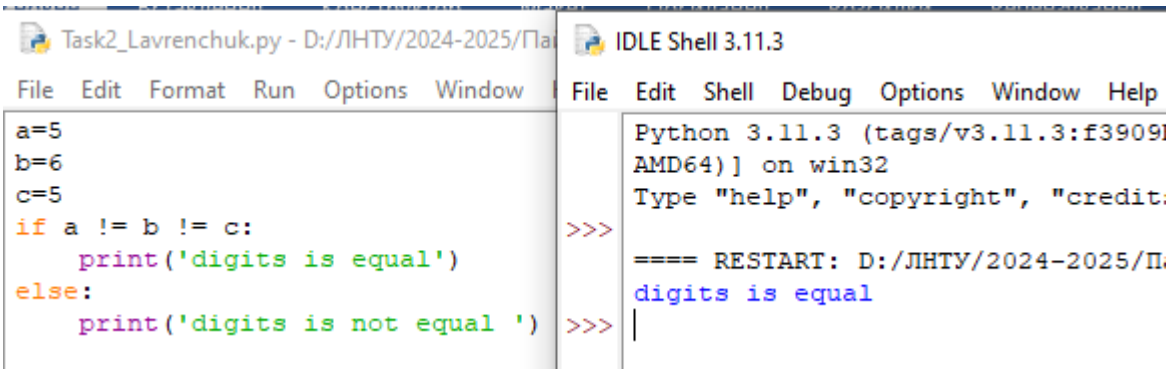


Рисунок 2.9 – Тестування транзитивності

2.2.5 Вкладені оператори if-else

Розглянемо два окремі випадки умовного оператора.

Спочатку розглянемо випадок, коли після інструкції `if` стоїть інша інструкція `if`.

Читайте, що ми запланували на цю Неділю. Якщо буде гарна погода, підемо гуляти. Якщо знайдемо гарний ресторан, то пообідаємо там. Інакше з’їмо бутерброд. Якщо буде погана погода, підемо в театр. Якщо не буде квитків, підемо за покупками в найближчий торговий центр.

Напишемо те ж саме на Python. Уважно подивіться на наведений тут код:

```
if the_weather_is_good:
    if nice_restaurant_is_found:
        have_lunch()
    else:
        eat_a_sandwich()
else:
    if tickets_are_available:
        go_to_the_theater()
    else:
        go_shopping()
```

Тут є два важливих моменти:

– таке використання оператора if відоме як *вкладеність*; пам'ятайте, що кожен else посилається на if, який знаходиться *на тому ж рівні відступу*; ви повинні знати це, щоб визначити, як if та else об'єднуються в пари;

– поміркуйте, як *відступи покращують читабельність* і роблять код легшим для розуміння та відстеження.

2.2.6 Оператор `elif`

Другий випадок використовує ще одне нове ключове слово мови Python: `elif`. Як ви, напевно, здогадуєтесь, це скорочена форма `else if`.

`elif` використовується для *перевірки більш ніж однієї умови* і для *зупинки*, коли буде знайдено перше вірне твердження.

Наш наступний приклад нагадує вкладення, але схожість дуже незначна. Знову ж таки, ми змінимо наші плани і висловимо їх наступним чином: Якщо буде гарна погода, підемо гуляти, інакше, якщо дістанемо квитки, підемо в театр, інакше, якщо будуть вільні столики в ресторані, підемо обідати, інакше залишимося вдома і будемо грати в шахи.

Ви помітили, скільки разів ми використали слово інакше? Саме на цьому етапі свою роль відіграє ключове слово `elif`.

Напишемо той самий сценарій за допомогою Python:

```
if the_weather_is_good:
    go_for_a_walk()
elif tickets_are_available:
    go_to_the_theater()
elif table_is_available:
    go_for_lunch()
```

```
else:
    play_chess_at_home()
```

Спосіб організації послідовних операторів `if-elif-else` іноді називають *каскадом*.

Зверніть увагу ще раз, як відступи покращують читабельність коду.

Тут слід звернути додаткову увагу на деякі моменти:

– ви не можете використовувати `else` без попереднього `if`;

– `else` завжди є останньою гілкою каскаду, незалежно від того, використовували ви `elif` чи ні;

– `else` є необов'язковою частиною каскаду і може бути опущена;

– якщо в каскаді є гілка `else`, то з усіх гілок виконується тільки одна;

– якщо немає гілки `else`, то можливо, що жодна з доступних гілок не виконується.

Примітка: якщо будь-яка з гілок `if-elif-else` містить лише одну інструкцію, то її можна кодувати в більш стислому вигляді (не потрібно робити відступ після ключового слова, а достатньо просто продовжити рядок після двокрапки):

```
if number1 > number2: larger_number = number1
else: larger_number = number2
```

Цей стиль, однак, може вводити в оману, і ми не рекомендуємо використовувати його, але про це, безумовно, варто знати, якщо ви хочете читати і розуміти чужі програми.

Розглянемо приклад – знайдемо найбільше з трьох чисел.

Спочатку ми припускаємо, що перше значення є найбільшим. Потім перевіримо цю гіпотезу з двома значеннями, що залишилися.

Це можна реалізувати за допомогою коду, наведеного в лістингу 2.1.

Лістинг 2.1 – Приклад використання операторів розгалуження

```
# Зчитайте три числа
number1 = int(input("Введіть перше число: "))
```

```
number2 = int(input("Введіть друге число: "))
number3 = int(input("Введіть третє число: "))

# Тимчасово припустимо, що перше число
# є найбільшим.
# Ми незабаром це перевіримо.
largest_number = number1

# Перевіряємо чи друге число більше поточного
# largest_number
# і оновлюємо largest_number, якщо потрібно.
if number2 > largest_number:
    largest_number = number2

# Перевіряємо, чи третє число більше поточного
# largest_number
# і оновлюємо largest_number, якщо потрібно.
if number3 > largest_number:
    largest_number = number3

# Виведіть результат
print("Найбільше число це:", largest_number)
```

Кінець лістингу 2.1

2.3 Логічні оператори

Часто в реальному житті ми погоджуємося або заперечуємо те чи інше твердження, подію, факт. Наприклад, «Сума чисел 3 та 5 більша за 7» є правдивим твердженням, а «Сума чисел 3 та 5 менша за 7» – хибним. Можна помітити, що з точки зору логіки подібні фрази припускають тільки два результати: «Так» (правда) і «Ні» (неправда). Подібне використовується в програмуванні: якщо результатом обчислення виразу може бути лише «Так» або «Ні», то такий вираз називається *логічним*. Як бути в ситуації, коли ми маємо кілька умов?

У Python є три логічні оператори, які дозволяють створювати складні умови:

– `and` – кон’юнкція або логічне множення;

– `or` – диз’юнкція або логічне додавання;

– `not` – логічне заперечення.

2.3.1 Оператор and

Припустимо, ми написали програму для учнів віком від дванадцяти років, які навчаються принаймні у 7 класі. Доступ до неї тим, хто молодший, треба заборонити. Наступний код вирішує поставлене завдання:

```
age = int(input('How old are you?: '))
grade = int(input('What year education do you have?: '))
if age >= 12 and grade >= 7:
    print('access is available.')
else:
    print('access denied.')
```

Ми об'єднали дві умови за допомогою оператора and. Воно означає, що в цьому розгалуженні блок коду виконується лише при виконанні обох умов *одночасно*!

Оператор and може об'єднувати довільну кількість умов:

```
age = int(input(' How old are you?: '))
grade = int(input(' What year education do you have?: '))
city = input('You hometown is?: ')
if age >= 12 and grade >= 7 and city == 'Lutsk':
    print('Access is available.')
else:
    print('Access denied.')
```

В таблиці 2.3 наведено таблицю істинності оператора and. У ній перераховані вирази, з'єднані оператором and, показані всі можливі комбінації істинності та хибності та наведені результуючі значення виразів.

Таблиця 2.3 – Таблиця істинності для оператора and

a	**b**	**a and b**
False	False	False
False	True	False
True	False	False
True	True	True

Як показує таблиця, щоб значення виразу з оператором and було істинним, мають бути істинними *обидві (усі)* об'єднані ним умови.

2.3.2 Оператор or

Оператор or також застосовується для об'єднання умов. Однак, на відміну від and, для виконання блоку коду достатньо виконання *хоча б однієї з умов*.

```
city = input(‘ You hometown is?: ‘)
if city == ‘Lutsk’ or city == ‘Uzhgorod’ or city == ‘Kyiv’:
    print(‘ Access is available.’)
else:
    print(‘ Access denied.’)
```

Доступ буде дозволено, якщо хоча б одна з умов виконається.

В таблиці 2.4 наведено таблицю істинності оператора or. У ній перераховані вирази, з'єднані оператором or, показані всі можливі комбінації істинності та хибності та наведені результуючі значення виразів.

Таблиця 3.4 – Таблиця істинності для оператора or

a	**b**	**a or b**
False	False	False
False	True	True
True	False	True
True	True	True

Для того, щоб вираз з or був істинним, потрібно, щоб *хоча б одна умова* оператора or була істинною. При цьому не має значення, істинним чи хибним є другий вираз.

Логічний вираз X and Y істинний, якщо обидва значення X та Y істинні.

Логічний вираз X or Y істинний, якщо хоча б одне із значень X та Y істинне. Ми можемо використовувати обидва логічні оператори одночасно:

```
age = int(input(‘How old are you?: ‘))
grade = int(input(‘ What year education do you have?: ‘))
city = input(‘You hometown is?: ‘)
if age >= 12 and grade >= 7 and ( city == ‘Lutsk’ or city ==
‘Uzhgorod’):
    print(‘Access is available.’)
else:
    print(‘Access denied.’)
```

Такий код перевіряє, що вік учнів від дванадцяти років і навчаються вони принаймні у 7 класі та живуть у Луцьку чи Ужгороді.

2.3.3 Оператор not

Оператор не дозволяє інвертувати (тобто замінити на протилежний) результат логічного виразу. Наприклад, наступний код:

```
age = int(input('How old are you?: '))
if not (age < 12):
    print('Access is available.')
else:
    print('Access denied.')
повністю еквівалентний коду:
age = int(input('How old are you?: '))
if age >= 12:
    print('Access is available.')
else:
    print('Access denied.')
```

У першому прикладі ми помістили вираз `age < 12` в дужки для того, щоб було чітко видно, що ми застосовуємо оператор `not` до значення виразу `age < 12`, а не тільки змінної `age`.

В таблиці 2.5 наведено таблицю істинності оператора `not`.

Таблиця 2.5 – Таблиця істинності для оператора not

a	**not a**
False	True
True	False

2.3.4 Пріоритети логічних операторів

Логічні оператори, подібно до арифметичних операторів `(+, -, *, /)`, мають пріоритет виконання. Пріоритет виконання наступний:

– в першу чергу виконується логічне заперечення `not`;

– далі виконується логічне множення `and`;

– далі виконується логічне додавання `or`.

Для явного вказівки порядку виконання умовних операторів використовують дужки.

Контрольні питання

1. Що таке оператори порівняння у Python і для чого вони використовуються?

2. Як працює оператор рівності `==` у Python? Наведіть приклад.
3. Для чого призначений оператор «не дорівнює» `!=`?
4. Які ще оператори порівняння існують у Python (крім `==` і `!=`)?
5. Як Python виконує порівняння числових і рядкових значень?
6. Що таке ланцюг порівнянь у Python? Наведіть приклад.
7. Як перевірити, чи певне число належить діапазону, використовуючи ланцюг порівнянь?
8. У чому полягає транзитивність порівнянь і як вона проявляється у Python?
9. Що таке умовне виконання програми? Як працює оператор `if`?
10. У чому різниця між операторами `if` та `if-else`?
11. Як створити вкладені умовні оператори у Python? Наведіть приклад.
12. Яке призначення оператора `elif` і чому він зручніший за вкладені `if`?
13. Які логічні оператори існують у Python? Поясніть роботу кожного з них.
14. Який порядок пріоритетів мають логічні оператори `and`, `or`, `not`?
15. Як можна комбінувати оператори порівняння та логічні оператори в одному виразі?

ТЕМА 3. Цикли

3.1 Створення циклів за допомогою оператора `while`

У реальному житті ми досить часто зустрічаємося з циклами. У комп'ютерних програмах поряд з інструкціями розгалуження (тобто вибором шляху дії) також існують інструкції циклів (повторення дії). Якби інструкцій циклу не існувало, довелося б багато разів вставляти в програму один і той же код поспіль стільки раз, скільки потрібно виконати однакову послідовність дій.

Цикли – це інструкції, які виконують одну й ту ж саму послідовність дій, поки діє задана умова.

Кожен циклічний оператор має тіло циклу – якийсь блок коду, який інтерпретатор буде повторювати поки умова повторення циклу буде залишатися істинною.

Цикл з конструкцією `while` є універсальним організатором циклу в мові програмування Python (як у багатьох інших мовах). Цей цикл має дві частини: умова, яка перевіряється на істинність чи хибність, та інструкцію чи набір інструкцій, які повторюються доти, доки умова є істинною. На рисунку 3.1 надана логічна схема циклу `while`.

Рисунок 3.1 – Логіка роботи циклу while

В узагальненому вигляді, у мові Python цикл можна представити наступним чином:

```
while
    instruction
```

Існує синтаксична подібність команд `if` та `while`, проте вони мають семантичну різницю:

– після перевірки умови `if` виконує свої оператори тільки один раз;

– `while` повторює виконання до тих пір, поки умова набуває значення `True`.

Примітка: всі правила, що стосуються відступів, діють і тут.

Розглянемо алгоритм:

```
while conditional_expression:
    instruction_one
    instruction_two
    instruction_three
    :
    :
    instruction_n
```

Важливо запам’ятати:

– якщо всередині одного циклу `while` потрібно виконати декілька інструкцій, то необхідно (як і у випадку з `if`) зробити однаковий відступ для всіх інструкцій;

– інструкція або набір інструкцій, що виконуються всередині циклу `while`, називається *тілом циклу*;

– якщо умова має значення `False` (дорівнює нулю) вже при першій перевірці, то тіло не виконується жодного разу;

– тіло повинно мати можливість змінювати значення умови, тому що якщо на початку умова була `True`, то тіло буде працювати безперервно до нескінченності.

Отож, під час досягнення кінця блоку коду команди `if`, виконання програми передається наступній команді, а під час досягнення кінця блоку коду команди `while`, керування передається на початок циклу і програма продовжує знову виконувати той самий блок коду (див. рисунок 3.1).

Розглянемо приклад циклу (рисунок 3.2).

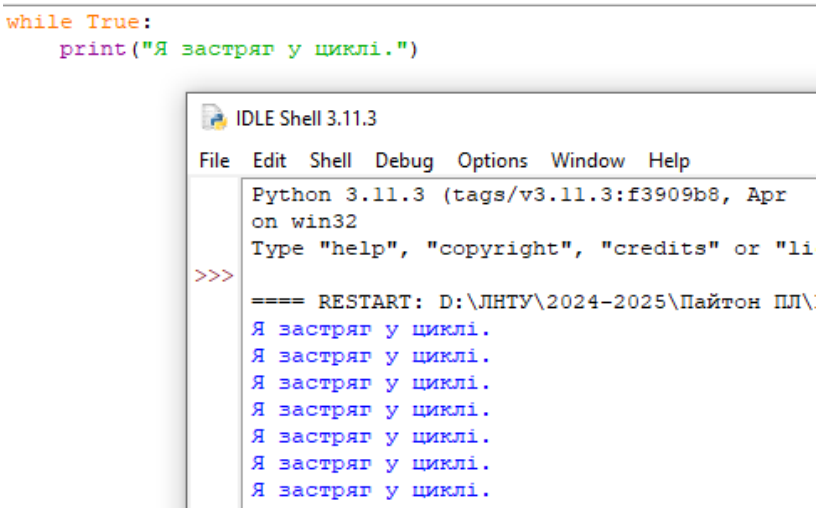


Рисунок 3.2 – Нескінченний цикл

Цей цикл буде нескінченно виводити на екран «Я застряг у циклі.». Щоб завершити роботу програми, просто натисніть Ctrl-C (або Ctrl-Break на деяких комп'ютерах). Це викличе переривання клавіатури і дозволить програмі вийти з циклу. Щоб уникнути випадкового виникнення нескінченних циклів потрібно слідувати щоб умова рано чи пізно могла перетворитися на `False`. Для цього зазвичай (часто, але не завжди) використовують додатковий параметр циклу.

Розглянемо ще один приклад:

```
i = 0  # початкове значення параметра циклу
while i < 10: # перевірка умови (тут задіяно параметр)
    print(i)
    i = i + 1 # зміна параметру (наближення умови до False)
```

Наданий код визначає, що доки змінна менше десяти, її потрібно виводити на екран. Далі збільшують її значення на одиницю. Якщо запустити цей код, він видасть значення від 0 до 9, кожна цифра буде в окремому рядку.

Примітка: у циклі while умова завжди перевіряється на початку кожної ітерації (ітерація – крок виконання циклу).

У прикладі на першому етапі присвоюється значення 0 змінній i. Цикл while порівнює значення змінної i з числом 10 та продовжує роботу, до тих пір, поки значення i є меншим 10 (умова циклу є істинною), у протилежному випадку цикл завершується (умова циклу є хибною).

Усередині циклу виводиться значення змінної i, що далі збільшується на 1. Мова програмування Python повертається до початку циклу і знову порівнює

значення змінної і з числом `10`. Значення змінної і на другому етапі дорівнює `1`, тому цикл `while` виконується знову і змінна і збільшується до `2`. Це триватиме поки значення змінної і у циклі не буде дорівнювати `10`. Під час чергового повернення на початок циклу перевірка `i < 10` поверне значення `False` і цикл `while` закінчується (`10` вже не виведе на екран). Мова програмування Python перейде до виконання наступних рядків коду (після циклу).

Таким чином, змінюючи значення змінної в тілі циклу, можна довести логічний вираз до хибності. Цю змінну, яка використовується в заголовку циклу `while`, зазвичай називають *лічильником*. Як і будь-якій змінній, їй дають довільні імена, часто використовуються букви `i` та `j`, можуть бути й інші імена.

В попередньому прикладі легко передбачити кількість кроків циклу (порахувати від 0 до 10). Розглянемо інший приклад, де кількість кроків залежить від користувача – поки він не введе -1 (рисунок 3.3).

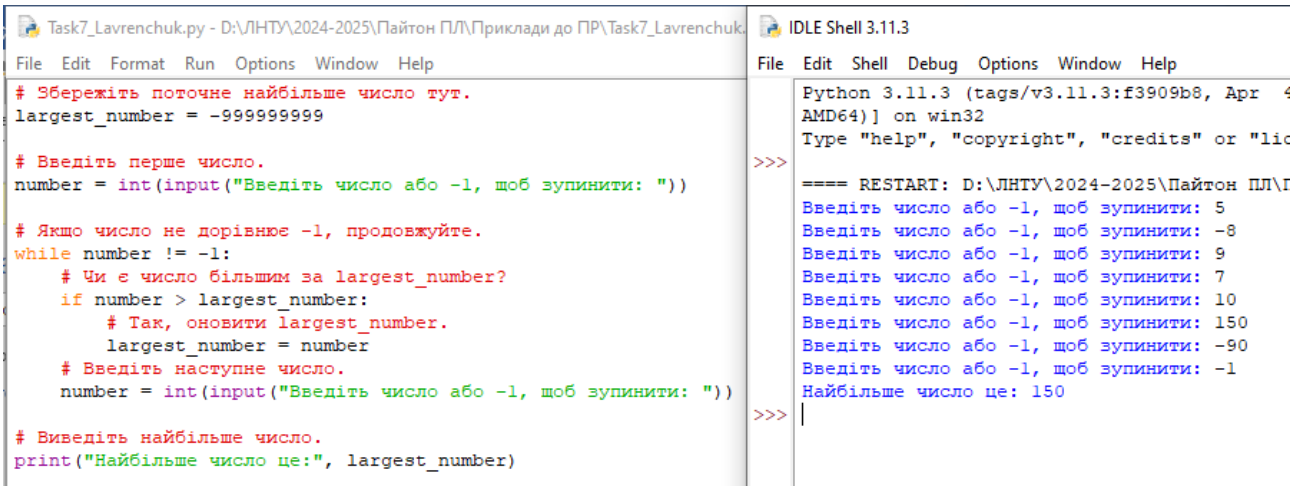


Рисунок 3.3 – Цикл з непередбачуваною кількістю ітерацій

Як видно з рисунку, в цій програмі поєднано використання циклу та перевірки умови. В тілі циклу перевіряється умова чи введене число є більшим за те, яке ми вважали найбільшим до цього. Якщо умова виконується, то ми тепер вважаємо щойно введене число максимальним, якщо ні, то максимальним лишається те, що було. У випадку використання таких циклів користувач сам не може здогадатися що йому потрібно ввести, щоб припинити роботу циклу, тому варто надати йому цю інформацію (вивести на екран відповідне повідомлення).

Розглянемо приклад на рисунку 3.4.

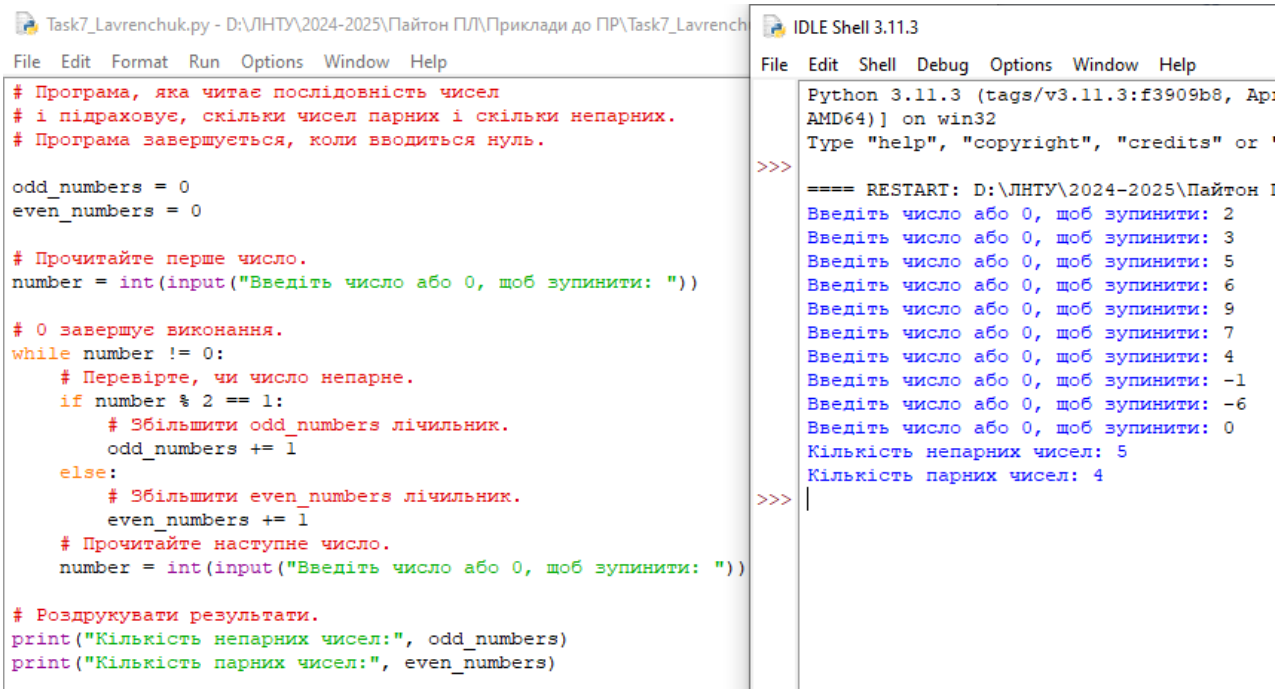


Рисунок 3.4 – Цикл та повна команда розгалуження

Деякі вирази можна спростити без зміни поведінки програми.

Спробуйте пригадати, як у мові Python інтерпретується істинність умови, і зверніть увагу, що ці дві форми еквівалентні:

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

та

```
while number:
```

У таких еквівалентних формах може бути закодована і умова, яка перевіряє, чи є число непарним:

```
if number % 2 == 1:
```

та

```
if number % 2:
```

Розглянемо приклад з рисунку 3.5.

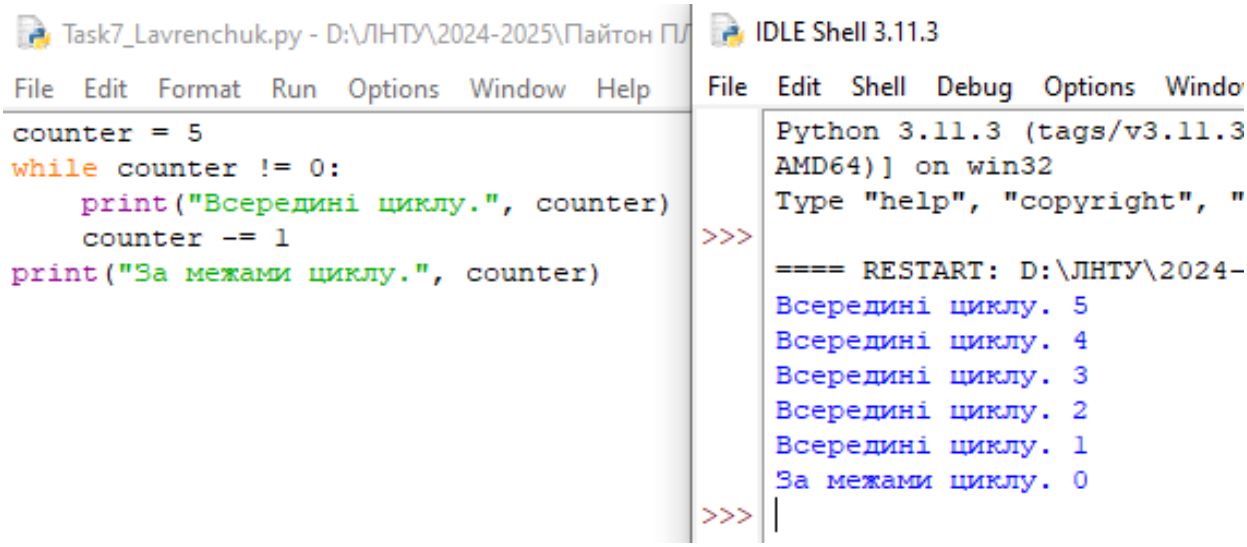


Рисунок 3.5 – Використання лічильника циклу

Даний код призначений для виведення рядка «Всередині циклу.» та значення, що зберігається в змінній `counter` протягом роботи заданого циклу рівно п’ять разів. Як тільки умова не виконана (значення змінної `counter` досягло 0), відбувається вихід з циклу, виводиться повідомлення «За межами циклу.», а також значення, що зберігається в `counter`.

Але форма умови циклу `while` можна написати компактніше:

```
while counter != 0:
```

замінити на:

```
while counter:
```

При цьому отримаємо точно такий самий результат, як на рисунку 3.5.

Приклад. Обчислити значення функції $y = \frac{x^3 - 4x + 1}{x + 1}$, користуючись оператором циклу `while`, при `x`, що змінюється в діапазоні `xпоч` ≤ `x` ≤ `xкін` з кроком `Δx` (розв’язок наведено на рисунку 3.6).

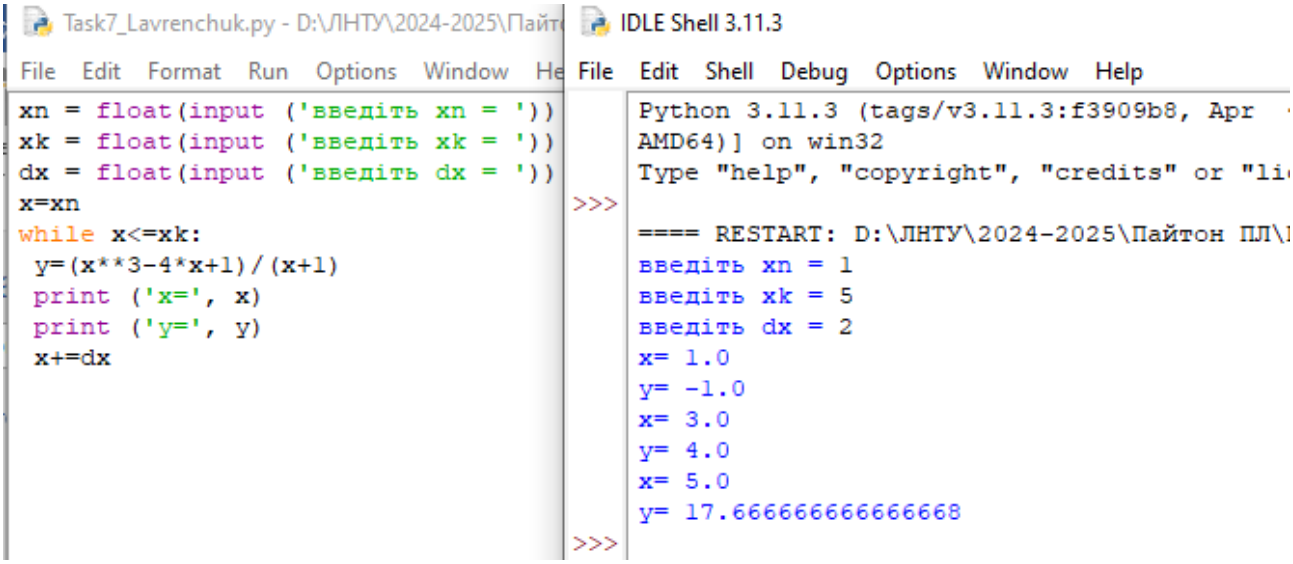


Рисунок 3.6 – Приклад використання циклу while

3.2 Створення циклів за допомогою оператора for

Ще один вид циклу, доступний в Python, походить від міркування, що іноді важливіше порахувати «проходи» циклу, ніж перевіряти умови.

Уявіть собі, що тіло циклу потрібно виконати рівно сто разів. Якщо для цього скористатися циклом `while`, то він буде мати такий вигляд:

```
i = 0
while i < 100:
    # do_something()
    i += 1
```

Аналогічні дії можна виконати за допомогою наступного коду:

```
for i in range(100):
    # do_something()
    pass
```

– ключове слово `for` відкриває цикл `for`; зверніть увагу – після нього немає жодної умови; вам не потрібно думати про умови, оскільки вони перевіряються всередині, без вашого втручання;

– будь-яка змінна після ключового слова `for` є *змінною керування* циклом, вона здійснює підрахунок тактів циклу, причому робить це автоматично;

– ключове слово `in` задає елемент синтаксису, що описує діапазон можливих значень, які присвоюються змінній керування;

– функція `range()` (це дуже особлива функція) відповідає за формування усіх потрібних значень для змінної керування, у нашому прикладі функція буде послідовно перебирати (можна навіть сказати, що буде передавати в цикл) значення з наступної послідовності: `0, 1, 2 .. 97, 98, 99`; примітка: у цьому випадку функція `range()` починає свою роботу з 0 і закінчує її за один крок (одне ціле число) до значення її аргументу;

– зверніть увагу на ключове слово `pass` – в тілі циклу воно взагалі нічого не робить, це порожня інструкція – ми помістили її сюди тому, що синтаксис циклу `for` вимагає наявності в тілі хоча б однієї інструкції (до речі – `if, elif, else` та `while` вимагають теж саме).

Розглянемо приклад, наведений на рисунку 3.7.

```
1  for i in range(3):
2   print("Значення і зараз дорівнює", i)
3
```

```
Console
Значення і зараз дорівнює 0
Значення і зараз дорівнює 1
Значення і зараз дорівнює 2
```

Рисунок 3.7 – Використання циклу `for`

Як бачимо цикл було виконано три рази (це аргумент функції `range()`) і останнє значення контрольної змінної дорівнює `2` (а не `3`, оскільки починається з `0`, а не з `1`).

Функція `range()` може приймати не один, а два аргументи. Розглянемо приклад з рисунку 3.8.

```
1  for i in range(2, 5):
2      print("Значення і зараз дорівнює", i)
3
```

```
Console
Значення і зараз дорівнює 2
Значення і зараз дорівнює 3
Значення і зараз дорівнює 4
```

Рисунок 3.8 – Функція `range()` з двома аргументами

В цьому випадку перший аргумент визначає початкове (перше) значення керуючої змінної.

Останній аргумент вказує на перше значення, яке *вже не буде* присвоєно керуючій змінній.

Примітка: функція range() приймає в якості аргументів *тільки цілі* числа, і генерує послідовності *лише цілих* чисел.

Як бачимо з рисунку 3.8, перше значення, що виводиться, дорівнює `2` (береться з першого аргументу `range()`), а останніє `4` (хоча другий аргумент в `range() 5`).

Функція `range()` також може приймати три аргументи (рисунок 3.9).

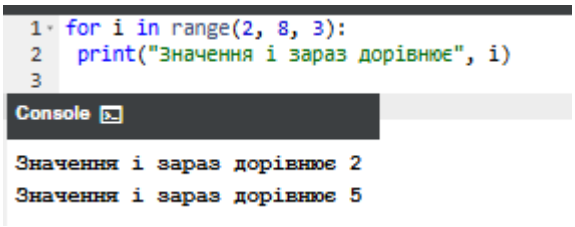


Рисунок 3.9 – Функція range() з трьома аргументами

Третім аргументом є *інкремент* – це значення, що додається для керування змінною при кожному повороті циклу (за замовчуванням значення інкременту дорівнює 1).

Перший аргумент, що передається у функцію `range()`, вказує на початковий номер послідовності (звідси `2` на виході).

Другий аргумент вказує функції, де закінчується послідовність (функція генерує числа, до числа вказаного в другому аргументі, але не включає його).

Третій аргумент вказує на крок, який фактично дорівнює різниці між сусідніми числами у згенерованій функцією послідовності чисел.

2 (початкове число) → 5 (2 збільшується на 3 і дорівнює 5 – число знаходиться в діапазоні від 2 до 8) → 8 (5 збільшується на 3 і дорівнює 8 – число не знаходиться в діапазоні від 2 до 8, оскільки значення аргументу stop не потрапляє в діапазон чисел, що генеруються функцією).

Примітка: якщо множина, сформована функцією `range()` порожня, то цикл взагалі не буде виконуватись (наприклад – рисунок 3.10).

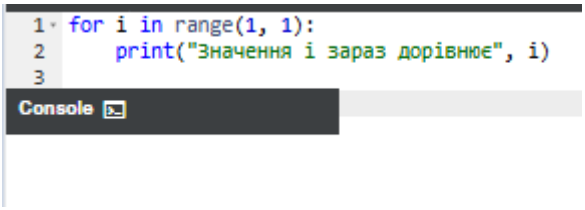


Рисунок 3.10 – Порожня множина в range()

Примітка: множина, згенерована функцією `range()`, повинна бути *відсортована за* зростанням. Неможливо змусити функцію `range()` створити множину в іншому вигляді, коли функція `range()` приймає саме два аргументи. Це означає, що другий аргумент `range()` повинен бути більшим за перший.

Таким чином, результату тут теж не буде:

```
for i in range(2, 1):
    print("Значення і зараз дорівнює", i)
```

Проте, коли використовуємо `range()` з трьома аргументами і останній аргумент від'ємний, то отримаємо *спадаючу послідовність* (рисунок 3.11).

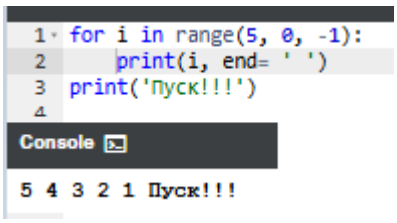


Рисунок 3.11 – Приклад використання від'ємного кроку

В загальному випадку, функція `range()` формує послідовність чисел. Вона приймає цілі числа, а повертає об'єкти з діапазону.

Синтаксис `range()` має наступний вигляд:

```
range(start, stop, step)
```

де:

– `start` – необов'язковий параметр, що задає початковий номер послідовності (за замовчуванням 0)

– `stop` – необов'язковий параметр, що вказує на кінець згенерованої послідовності (не входить до складу),

– `step` – необов'язковий параметр, що задає крок інтервалу між числами в послідовності (за замовчуванням – 1).

Приклади використання функції range() наведено в таблиці 3.1.

Таблиця 3.1 – Приклади використання функції range()

Виклик функції	***Послідовність чисел***
`range(10)`	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
`range(1, 10)`	1, 2, 3, 4, 5, 6, 7, 8, 9
`range(3, 7)`	3, 4, 5, 6
`range(7, 3)`	порожня послідовність
`range(2, 15, 3)`	2, 5, 8, 11, 14
`range(9, 2, -1)`	9, 8, 7, 6, 5, 4, 3
`range(3, 10, -2)`	порожня послідовність

Розглянемо коротку програму, метою якої є обчислення декількох перших степенів двійки (рисунок 3.12).

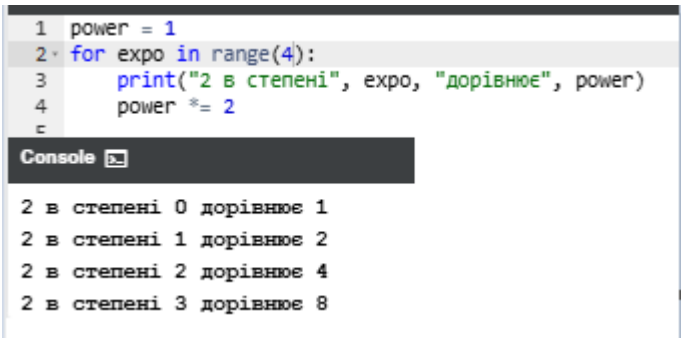


Рисунок 3.12 – Приклад використання циклу for

Змінна `expo` використовується як керуюча змінна циклу і вказує на поточне значення степеня. Саме піднесення до степеня замінюється множенням на два. Оскільки 2^0 дорівнює 1, то 2×1 дорівнює 2^1, 2×2^1 дорівнює 2^2 і так далі.

3.3 Оператори break та continue

Досі ми розглядали тіло циклу як неподільну і нерозривну послідовність інструкцій, які виконуються повністю на кожному проході циклу. Однак, як розробник, ви можете зіткнутися з наступним розвитком подій:

– якщо виявляється, що немає необхідності продовжувати цикл далі, а слід припинити подальше виконання тіла циклу і перейти далі;

– виникає необхідність почати наступний оберт циклу, не завершивши виконання поточного.

Python пропонує дві спеціальні інструкції для втілення обох цих задач. Для повноти картини скажемо, що їх наявність у мові не є обов'язковою – досвідчений програміст здатен закодувати будь-який алгоритм і без цих інструкцій. Такі доповнення, які не покращують виразності мови, а лише спрощують роботу розробника, іноді називають синтаксичними цукерками, або синтаксичним цукром.

Ці дві інструкції:

– `break` – негайний вихід з циклу, безумовне завершення роботи циклу; програма починає виконувати найближчу інструкцію після тіла циклу;

– `continue` – поводиться так, ніби програма раптово дійшла до кінця тіла; починається виконання наступного оберту і негайно перевіряється умова.

Обидва ці слова є ключовими.

Розглянемо два простих приклади, які ілюструють, як працюють ці дві інструкції (рисунок 3.13).

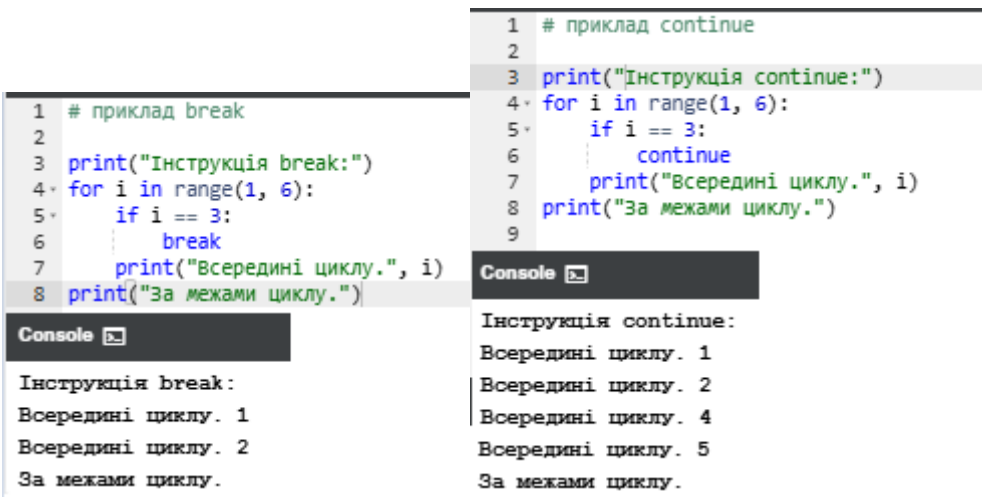


Рисунок 3.13 – Приклади використання break та continue

3.4 Поєднання циклів та розгалужень

Мова Python дозволяє використовувати розширений варіант оператора циклу – цикли можуть мати гілку `else`, подібно до `if`.

Гілка `else` в циклі завжди виконується один раз, незалежно від того, чи входив цикл в своє тіло чи ні.

Поки виконується умова повторення тіла циклу, оператор while працює так само, як і в звичайному варіанті, але як тільки умова повторення перестає виконуватися, потік виконання направляється по альтернативній гілці else – так само, як в умовному операторі if, вона виконається всього один раз (рисунок 3.14).

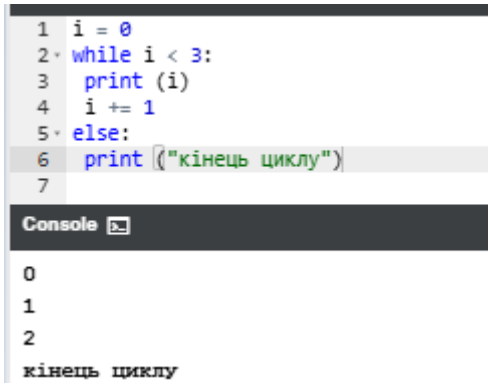


Рисунок 3.14 – Приклади використання while та else

Розглянемо приклад, наведений на рисунку 3.15.

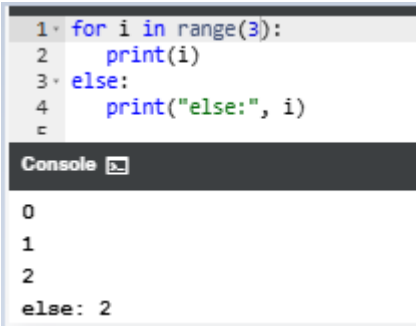


Рисунок 3.15 – Приклади використання for та else

Як бачимо, змінна `i` зберігає своє останнє значення, а якщо тіло циклу не виконуватиметься зовсім, керуюча змінна зберігатиме значення, яке вона мала до початку циклу.

Контрольні питання

1. Що таке цикл у програмуванні і для чого він використовується?
2. Як створюється цикл за допомогою оператора `while` у Python?

3. Що таке *нескінченний цикл* і як його уникнути?
4. Яка структура оператора `while`? Наведіть приклад простого циклу.
5. У чому полягає різниця між циклами `while` та `for`?
6. Як створити цикл за допомогою оператора `for` у Python?
7. Яку роль відіграє функція `range()` у циклах `for`?
8. Які параметри може мати функція `range()` і як вони впливають на роботу циклу?
9. Як виконати цикл із кроком, відмінним від 1, за допомогою `range()`?
10. Для чого використовується оператор `break` у циклах? Наведіть приклад.
11. Як працює оператор `continue` і чим він відрізняється від `break`?
12. Як можна поєднувати цикли з умовними операторами `if`, `elif`, `else`?
13. Що станеться, якщо в тілі циклу забути змінювати лічильник умови (`while`)?
14. Як можна використовувати вкладені цикли у Python? Наведіть приклад.
15. Як завершити виконання циклу достроково за певної умови?

ТЕМА 4. Списки. Сортування списків

4.1 Основи роботи зі списками.

Поки що ми навчилися оголошувати змінні, здатні зберігати лише одне значення за раз. Такі змінні іноді називають *скалярами* за аналогією з математикою. Усі змінні, які ви використовували досі, насправді є скалярами.

Масив – набір фіксованої кількості елементів, що розміщені в пам'яті комп'ютера безпосередньо один за одним, а доступ до них здійснюється за індексом (номер даного елементу в масиві).

В Python для реалізації масиву використовуються списки. *Список* – тип даних, що представляє собою послідовність певних значень, що можуть повторюватись. Але на відміну від масиву – кількість елементів у списку може бути довільною.

Списки – гетерогенна, змінювана структура даних, що може містити елементи різних типів, що перераховані через кому та заключені в квадратні дужки. Це дозволяє створювати структури будь-якої складності і глибини.

Списки служать для того, щоб зберігати об'єкти в певному порядку, особливо якщо порядок або вміст можуть змінюватися. Можна змінювати список, додати в нього нові елементи, а також видалити або перезаписати існуючі. Можна змінити кількість елементів у списку, а також самі елементи. Одне і те ж значення може зустрічатися в списку кілька разів.

Список – це тип даних у мові Python, який використовується для зберігання декількох об'єктів. Це впорядкований і змінний набір елементів, розділених комами, в квадратних дужках.

Створимо змінну з назвою `numbers`, якій присвоюється не одне число, а список з п'яти значень (*зверніть увагу*: список починається з відкритої квадратної дужки і закінчується закритою квадратною дужкою; простір між дужками заповнюється п'ятьма числами, розділеними комами).

```
numbers = [10, 5, 7, 2, 1]
```

`numbers` – це список, що складається з п’яти значень, і всі вони – числа. Також можна сказати, що цей оператор створює список довжиною що дорівнює п’яти (оскільки всередині нього знаходиться п’ять елементів).

Список є об’єктом, тому може бути присвоєний змінній. Список можна створити з нуля або більше елементів, розділених комами і вкладених у квадратні дужки:

```
empty_list = [ ]
number_list = [1, 2, 3, 4, 5]
week_days = [‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’, ‘Friday’]
print(empty_list)
print(number_list)
print(week_days)
```

У мові Python прийнято вважати, що елементи списку *завжди нумеруються, починаючи з нуля*. Це означає, що елемент, який зберігається на початку списку, матиме нульовий номер. Оскільки в нашому списку п’ять елементів, останньому з них присвоєно номер чотири. Не забувайте про це.

Список є набором елементів, але кожен елемент є скаляром.

4.2 Індексація списків

Присвоїмо першому елементу списку нове значення `111`. Зробимо це таким чином:

```
numbers = [10, 5, 7, 2, 1]
# Виведення початкового вмісту списку:
print("Оригінальний вміст списку:", numbers)
numbers[0] = 111
# Поточний вміст списку:
print("Новий вміст списку: ", numbers)
```

Значення п’ятого елемента скопіюємо до другого елемента:

```
numbers = [10, 5, 7, 2, 1]
# Виведення початкового вмісту списку:
print("Оригінальний вміст списку:", numbers)
numbers[0] = 111
```

```
# Виведення попереднього вмісту списку:
print("\nВміст попереднього списку:", numbers)
# Копіювання значення п’ятого елемента в другий:
numbers[1] = numbers[4]
# Виведення поточного вмісту списку:
print("Новий вміст списку:", numbers)
```

Значення в дужках, яке вибирає один елемент списку, називається *індексом*, тоді як операція вибору елемента зі списку відома як *індексування*.

Ми будемо використовувати функцію print() для виведення вмісту списку при кожному внесенні змін. Це допоможе нам більш ретельно відслідковувати кожен крок і бачити, що відбувається після тієї чи іншої модифікації списку.

Примітка: всі індекси, що використовувалися до цього часу, були цифрами. Їх значення фіксовані під час виконання, але індексом може бути будь-який вираз. Це відкриває багато можливостей.

4.3 Доступ до вмісту списку

До кожного з елементів списку можна отримати доступ окремо. Наприклад, його можна вивести на екран:

```
print(numbers[0]) # Доступ до першого елемента списку.
```

Список також можна вивести цілком:

```
print(numbers)  # Друк всього списку.
```

Довжина списку може змінюватися під час роботи з ним. До списку можуть бути додані нові елементи, а деякі – вилучені. Це говорить про те, що список є дуже динамічною конструкцією.

Якщо потрібно перевірити поточну довжину списку, то можна скористатися функцією `len()` (її назва походить від length – довжина).

Функція приймає в якості аргументу ім’я списку, а повертає кількість елементів, що зберігається в списку на даний момент (іншими словами – довжину списку).

Розглянемо приклад, наведений на рисунку 4.1.

```
1  numbers = [10, 5, 7, 2, 1]
2  print("Початковий вміст списку:", numbers) # Виведення початкового вмісту списку.
3
4  numbers[0] = 111
5  print("\nПопередній вміст списку:", numbers) # Виведення попереднього вмісту списку.
6
7  numbers[1] = numbers[4] # Копіювання значення п'ятого елементу до другого.
8  print("Попередній вміст списку:", numbers) # Виведення попереднього вмісту списку.
9
10 print("\nДовжина списку:", len(numbers)) # Виведення довжини списку.
11
```

Console

```
Початковий вміст списку: [10, 5, 7, 2, 1]

Попередній вміст списку: [111, 5, 7, 2, 1]
Попередній вміст списку: [111, 1, 7, 2, 1]

Довжина списку: 5
```

Рисунок 4.1 – Приклад доступу до елементів списку

Зверніть увагу, що довжина списку – кількість елементів в ньому, вона на одиницю більша, ніж індекс останнього елементу.

4.4 Видалення елементів зі списку

Будь-який з елементів списку можна *видалити* в будь-який момент – це робиться за допомогою команди `del` (видалити, англ. delete). Примітка: це *інструкція*, а не функція, тому *біля її імені немає дужок*.

Потрібно вказати на елемент, який потрібно видалити – він зникне зі списку, а довжина списку зменшиться на одиницю, наприклад:

```
del numbers[4]
```

Ви не можете отримати доступ до елементу, якого не існує – ви не можете ні отримати його значення, ні присвоїти йому значення. Обидві інструкції тепер викликатимуть *помилки* під час виконання:

```
print(numbers[4])
numbers[4] = 1
```

Розглянемо приклад, наведений на рисунку 4.2.

```
numbers = [10, 5, 7, 2, 1]
print("Початковий вміст списку:", numbers) # Виведення початкового вмісту списку.

numbers[0] = 111
print("\nПопередній вміст списку:", numbers) # Виведення попереднього вмісту списку.

numbers[1] = numbers[4] # Копіювання значення п'ятого елементу до другого.
print("Попередній вміст списку:", numbers) # Виведення попереднього вмісту списку.

print("\nДовжина списку:", len(numbers)) # Виведення попередньої довжини списку.

###

del numbers[4] # Видалення другого елементу зі списку.
print("Нова довжина списку", len(numbers)) # Виведення довжини нового списку.
print("\nНовий вміст списку:", numbers) # Виведення поточного вмісту списку.

###
```

Console

```
Початковий вміст списку: [10, 5, 7, 2, 1]

Попередній вміст списку: [111, 5, 7, 2, 1]
Попередній вміст списку: [111, 1, 7, 2, 1]

Довжина списку: 5
Нова довжина списку 4

Новий вміст списку: [111, 1, 7, 2]
```

Рисунок 4.2 – Приклад використання команди `del`

Як видно з рисунку 4.2, ми вилучили один з елементів списку – тепер у списку лише чотири елементи. Це означає, що елемента номер чотири не існує.

```
del my_list  # видаляє весь список
```

4.5 Від'ємні індекси

Можна використовувати від'ємні індекси при роботі зі списками, при цьому Елемент з індексом `-1`, є останнім у списку, елемент з індексом -2 буде передостаннім у списку (рисунок 4.3).

```
numbers = [111, 7, 2, 1]
print(numbers[-1])
```

Console

```
1
```

```
numbers = [111, 7, 2, 1]
print(numbers[-2])
```

Console

```
2
```

Рисунок 4.3 – Приклад використання від'ємних індексів

Зрозуміло, що для наведеного прикладу списку останнім доступним елементом у буде `numbers[-4]`, вихід за межі кількості елементів у списку призведе до помилки.

Задача 1. Є частинка групи з перших п'яти студентів: `1, 2, 3, 4` і `5`, яка зберігається в списку. Всі, хто є в списку, по черзі мають вийти до дошки. Реалізувати сценарій:

–Крок 1. Середній студент в списку не хоче йти до дошки і пропонує замість себе когось іншого (написати рядок коду, який пропонує користувачу замінити середнє число у списку на ціле число, введене користувачем).

–Крок 2. Останній студент в списку вийшов з аудиторії, відповідно його неможливо викликати до дошки (написати рядок коду, який видаляє останній елемент зі списку).

–Крок 3. Порахувати скільки студентів вийде до дошки на занятті та вивести їх список (написати рядок коду, що виводить довжину списку та сам модифікований список).

Розв'язок цієї задачі наведено на рисунку 4.4. Зверніть увагу на рядок 5, функція перетворення типів `int()` використана для того, щоб всі елементи в списку були цілого типу. Якщо її опустити:

```
x=(input("введіть ціле число"))
```

то отримаємо результат – список змішаного типу, тобто введене нами число буде сприйматися як символ:

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

```
1  group_list = [1, 2, 3, 4, 5] # Це початковий список студентів
2
3  # Крок 1: написати рядок коду, який пропонує користувачеві
4  # замінити середнє число на ціле число, введене користувачем.
5  x=int(input("введіть ціле число"))
6  group_list[2]=x
7  print(group_list)
8  # Крок 2: написати рядок коду, який видаляє останній елемент зі списку.
9  del group_list[4]
10 # Крок 3: написати рядок коду, що виводить довжину списку.
11 print(len(group_list))
12 print(group_list)
13
14
Console
введіть ціле число6
[1, 2, 6, 4, 5]
4
[1, 2, 6, 4]
```

Рисунок 4.4 – Розв'язок задачі про студентів

4.6 Функції та методи роботи зі списками

Метод – це специфічний тип функції, який поводиться як функція і виглядає як функція, але відрізняється від неї способом дії та стилем виклику.

Функція не належить жодним даним – вона отримує дані, вона може створювати нові дані і вона (як правило) видає результат.

Метод робить усе те саме, але ще й здатен змінювати стан вибраного об'єкта.

Метод належить даним, з якими він працює, в той час як функція належить всьому коду.

Це також означає, що виклик методу вимагає певної специфікації даних, з яких він викликається.

Це може здатися незрозумілим, але ми розберемося з цим більш детально, коли заглибимося в об'єктно-орієнтоване програмування.

Стандартний виклик функції може виглядати таким чином:

```
result = function(arg)
```

Функція отримує аргумент, щось робить і повертає результат.

Типовий виклик методу зазвичай виглядає так:

```
result = data.method(arg)
```

Примітка: перед назвою методу ставиться назва даних, які належать цьому методу. Далі ви ставите *крапку*, за якою йде *ім'я методу*, і пара *круглих дужок*, що містять аргументи.

Метод буде вести себе як функція, але може робити дещо більше – він може змінювати внутрішній зміст даних, з яких він був викликаний.

Новий елемент можна приклеїти до кінця існуючого списку за допомогою методу `append()`:

```
list.append(value)
```

Він приймає значення аргументу та розміщує його в кінці списку, на який вказує метод. Після цього довжина списку збільшується на одиницю.

Метод `insert()` трохи гнучкіший – він може додати новий елемент в будь-яке місце списку, а не тільки в кінець.

```
list.insert(location, value)
```

Для роботи з цим методом потрібно два аргументи:

–перший вказує на необхідну позицію вставки елемента; примітка: всі існуючі елементи, які займають місця праворуч від нового елемента (в тому числі і той, що знаходиться на вказаній позиції), зміщуються вправо, для того, щоб звільнити місце для нового елемента;

–другий – це елемент, який потрібно вставити.

Розглянемо приклад, наведений на рисунку 4.5.

```
1  numbers = [111, 7, 2, 1]
2  print(len(numbers))
3  print(numbers)
4
5  ###
6
7  numbers.append(4)
8
9  print(len(numbers))
10 print(numbers)
11
12 ###
13
14 numbers.insert(0, 222)
15 print(len(numbers))
16 print(numbers)
17
```

Console

```
4
[111, 7, 2, 1]
5
[111, 7, 2, 1, 4]
6
[222, 111, 7, 2, 1, 4]
```

Рисунок 4.5 – Приклад використання методів `append()` та `insert()`

Як бачимо з рисунку 4.5, після використання функції `insert()`: колишній перший елемент стає другим, другий – третім і так далі.

Ми можемо спочатку створити список порожнім (це робиться за допомогою порожньої пари квадратних дужок), а потім додавати до нього нові елементи за потребою. Розглянемо приклад, наведений на рисунку 4.6.

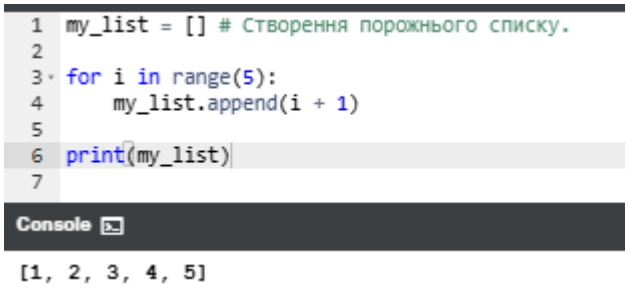


Рисунок 4.6 – Приклад створення порожнього списку та його заповнення

Ми отримали послідовність цілих чисел, починаючи з 1 (початкове значення змінної `i` 0, ми додаємо одиницю до всіх доданих значень і рухаємося до 5). В прикладі, наведеному на рисунку 4.7, відбуваються ті самі дії, але щоразу елементи додаються не в кінець списку, а на його початок (в позицію 0).

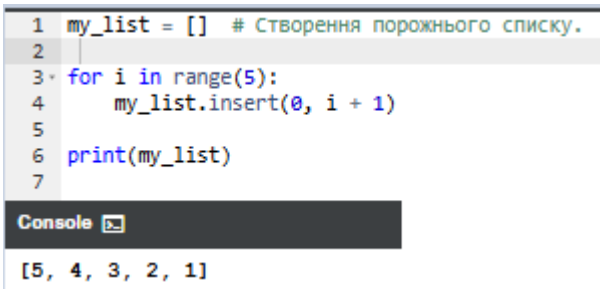


Рисунок 4.7 – Приклад заповнення списку у зворотному порядку

4.7 Використання списків

Нехай нам потрібно обчислити суму всіх значень, що зберігаються в списку `my_list`.

Потрібна змінна, в якій буде зберігатися сума і початкове значення якої буде дорівнювати 0 – назвемо її `total`. (*Примітка*: не будемо називати її `sum`, оскільки Python використовує таку ж назву для однієї з своїх вбудованих функцій: `sum()`. Використання однієї і тієї ж назви, як правило, вважається поганою практикою.) Потім додамо до неї всі елементи списку використовуючи цикл `for` (рисунок 4.8).

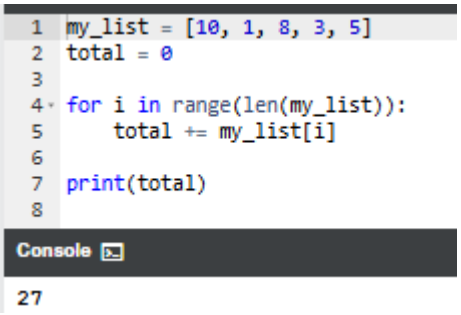


Рисунок 4.8 – Приклад знаходження суми елементів списку. I спосіб

Прокоментуємо цей приклад:

–список містить послідовність з п’яти цілочисельних значень;

–змінна `i` приймає значення `0, 1, 2, 3` та `4`, потім індексує список, вибираючи наступний елемент: перший, другий, третій, четвертий і п’ятий;

–кожен з цих елементів додається оператором `+=` до змінної `total`, яка містить кінцевий результат по закінченню циклу;

–зверніть увагу, як використовується функція `len()` – це робить код незалежним від будь-яких можливих змін у вмісті списку.

Розглянемо ще один приклад знаходження суми елементів списку (рисунок 4.9).

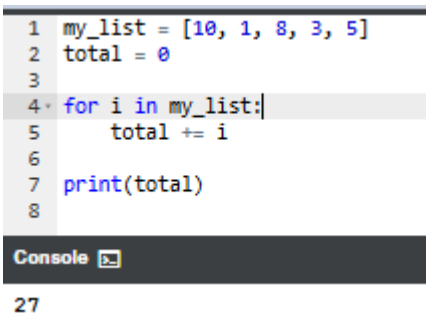


Рисунок 4.9 – Приклад знаходження суми елементів списку. II спосіб

Прокоментуємо що відбувається в коді, наведеному на рисунку 4.9:

–в інструкції `for` вказується змінна, яка використовується для перебору списку (тут `i`), за якою слідує ключове слово `in` та ім’я списку, який перебирається (тут `my_list`);

–змінній `i` присвоюються значення усіх наступних елементів списку, і процес повторюється стільки разів, скільки елементів у списку;

–це означає, що ми використовуємо змінну `i` *як копію значень елементів*, і нам не потрібно використовувати індекси;

–функція `len()` тут також не потрібна.

Отже, списки можна *перебирати* за допомогою циклу `for`, наприклад:

```
my_list = ["білий", "пурпурний", "блакитний", "жовтий", "зелений"]

for color in my_list:
    print(color)
```

4.8 Перестановки елементів списку

Нехай нам потрібно поміняти місцями значення двох змінних. Якщо ми напишемо код:

```
variable_1 = 1
variable_2 = 2
variable_2 = variable_1
variable_1 = variable_2
```

то втратимо значення, яке раніше зберігалось у `variable_2`. Зміна порядку виконання команд не допоможе. Потрібна третя змінна, яка виконує функцію додаткового сховища.

Це можна зробити наступним чином:

```
variable_1 = 1
variable_2 = 2
auxiliary = variable_1
variable_1 = variable_2
variable_2 = auxiliary
```

Проте у Python є зручніший спосіб зробити обмін:

```
variable_1 = 1
variable_2 = 2
variable_1, variable_2 = variable_2, variable_1
```

Аналогічно можна поміняти місцями й елементи списку:

```
my_list = [10, 1, 8, 3, 5]
my_list[0], my_list[4] = my_list[4], my_list[0]
my_list[1], my_list[3] = my_list[3], my_list[1]
print(my_list)
```

Результатом виконання останнього коду буде:

```
[5, 3, 8, 1, 10]
```

Розвинемо цю ідею для списку не з 5 елементів, а довільної довжини. Тут знадобиться оператор циклу `for`:

```
for i in range(length // 2):
    my_list[i], my_list[length - i - 1] = my_list[length - i - 1], my_list[i]
print(my_list)
```

Пояснення до цього коду:

–присвоюємо змінній `length` довжину поточного списку (це робить наш код трохи зрозумілішим та коротшим);

–запускаємо цикл `for` для проходу по його тілу `length // 2` разів (це добре працює для списків як парної, так і непарної довжини, тому що коли список містить непарну кількість елементів, то середній елемент залишається недоторканим);

–ми поміняли місцями `i`-й елемент (з початку списку) на елемент з індексом, що дорівнює (`length - i - 1`) (з кінця списку); в нашому прикладі коли `i` дорівнює `0`, то (`length - i - 1`) дає `4`; коли `i` дорівнює `1`, то дає `3`.

4.9 Сортування списків

На сьогоднішній день відомо багато алгоритмів сортування, які дуже відрізняються як за швидкістю роботи, так і за складністю. Розглянемо дуже простий алгоритм, який легко зрозуміти, але, на жаль, і він не надто ефективний. Він використовується дуже рідко, і звісно, не для великих і розгалужених списків.

Список можна сортувати двома способами:

–зростаючим (а точніше – неспадаючим) – якщо в кожній парі сусідніх елементів перший елемент не більший за другий;

–спадаючим (точніше – незростаючим) – якщо в кожній парі сусідніх елементів перший елемент не менший за другий.

Відсортуємо список за зростанням, тобто числа будуть впорядковані від найменшого до найбільшого.

Нехай початковий список такий:

8	10	6	2	4

Ми спробуємо використати наступний підхід: візьмемо перший і другий елементи і порівняємо їх; якщо виявиться, що вони стоять в хибній послідовності (тобто перший більший за другий), поміняємо їх місцями; якщо послідовність вірна, то нічого не робитимемо. Поглянувши на наш список, можна переконатися, що елементи перші два елементи розташовані в правильній послідовності, так як `8 < 10.`

Тепер подивимося на другий і третій елементи. Вони розташовані не в тій послідовності. Нам треба поміняти їх місцями:

8	10	6	2	4

→

8	6	10	2	4

Рухаємося далі і дивимося на третій і четвертий елементи. Знову ж таки, повинно бути не так. Нам треба поміняти їх місцями:

8	6	10	2	4

→

8	6	2	10	4

Тепер перевіряємо четвертий і п’ятий елементи. Так, вони теж не на тих позиціях. Відбувається ще один обмін:

8	6	2	10	4

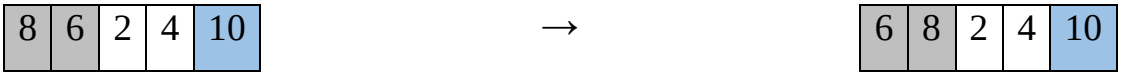
→

8	6	2	4	10

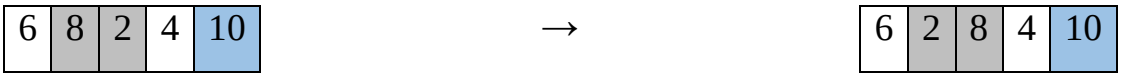
Перший прохід по списку вже завершено. Ми ще далекі від завершення справи, але тим часом сталося дещо цікаве. Найбільший елемент 10 вже «добіг» кінця списку. Зазначимо, що це саме те *місце, яке для нього призначене*.

Можна сказати, що це число спливло з дна в кінець списку, як бульбашка в келиху шампанського. Цей метод сортування отримав свою назву від того ж самого спостереження – його назвали сортуванням методом бульбашки.

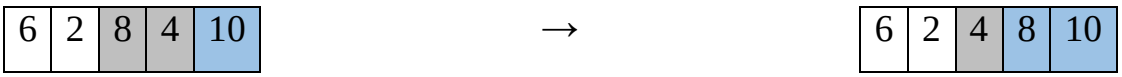
Тепер почнемо другий прохід по списку. Дивимось, що перший і другий елементи потребують перестановки:



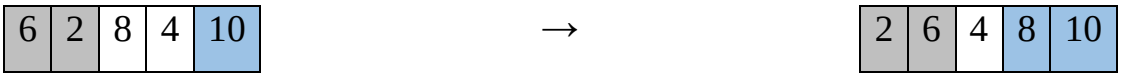
Другий і третій елементи теж потрібно поміняти місцями:



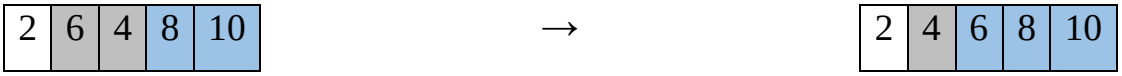
Третій і четвертий елементи також треба поміняти, після цього другий прохід завершується, оскільки 8 вже стоїть на своєму місці:



Одразу починаємо наступний прохід. Уважно дивіться на перший і другий елементи – потрібна ще одна перестановка:



Залишається ще 6, яка має стати на своє місце. Міняємо місцями другий і третій елементи:



Отримали впорядкований список. Нам більше нічого тут робити.

Як бачите, суть цього алгоритму проста: ми порівнюємо сусідні елементи, і помінявши місцями деякі з них, досягаємо своєї мети.

Алгоритм сортування бульбашкою можна також подивитися за посиланням: https://www.youtube.com/watch?v=Iv3vgjM8Pv4

Спробуймо реалізувати цей алгоритм за допомогою мови програмування Python:

```
my_list = [8, 10, 6, 2, 4]  # список для сортування
# нам потрібно (5 - 1) порівнянь
for i in range(len(my_list) - 1):
# порівняти сусідні елементи
    if my_list[i] > my_list[i + 1]:
        my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]
# Якщо ми опинимося тут, нам доведеться
            # поміняти елементи місцями
```

Описаний вище алгоритм потрібен для одного проходу, але ми бачили вище що за перший прохід масив ще лишається невідсортованим.

Ми вирішуємо це питання таким чином: *вводимо ще одну змінну*, завдання якої – спостерігати, чи відбулася перестановка при проходженні, чи ні; якщо перестановки немає, то список вже відсортований, і більше нічого робити не

треба. Створюємо змінну з іменем `swapped`, і присвоюємо їй значення `False`, щоб зазначити, що обмінів немає. В іншому випадку їй буде присвоєно значення `True` (рисунок 4.10).

```
my_list = [8, 10, 6, 2, 4]  # list to sort
swapped = True  # Це невеличка хитрість, щоб увійти в цикл while.

while swapped:
    swapped = False  # поки немає обмінів
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True  # відбувся обмін!
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]

print(my_list)
```

```
Console
[2, 4, 6, 8, 10]
```

Рисунок 4.10 – Приклад сортування списку методом бульбашки

Узагальнимо цю програму для довільного списку, введеного з клавіатури (рисунок 4.11).

```
my_list = []
swapped = True
num = int(input("Скільки елементів потрібно відсортувати: "))

for i in range(num):
    val = float(input("Введіть елемент списку: "))
    my_list.append(val)

while swapped:
    swapped = False
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]

print("\nВідсортовано:")
print(my_list)
```

```
Console
Скільки елементів потрібно відсортувати: 5
Введіть елемент списку: 4
Введіть елемент списку: 6
Введіть елемент списку: -7
Введіть елемент списку: 2
Введіть елемент списку: 5

Відсортовано:
[-7.0, 2.0, 4.0, 5.0, 6.0]
```

Рисунок 4.11 – Інтерактивна версія сортування списку методом бульбашки

Python має власні механізми сортування. Не потрібно писати свої власні сортування, оскільки є достатня кількість готових інструментів.

Ми розглянули цей алгоритм сортування, тому що важливо навчитися обробляти вміст списків, а також показати, як може працювати реальне сортування.

Якщо ви хочете, щоб Python відсортував ваш список, ви можете зробити це так:

```
my_list = [8, 10, 6, 2, 4]
my_list.sort()
print(my_list)
```

Це дуже просто.

Результат роботи фрагменту виглядає наступним чином:

```
[2, 4, 6, 8, 10]
```

Як бачите, всі списки мають метод `sort()`, який сортує їх максимально швидко.

Подібно можна сортувати списки різних типів (рисунок 4.12).

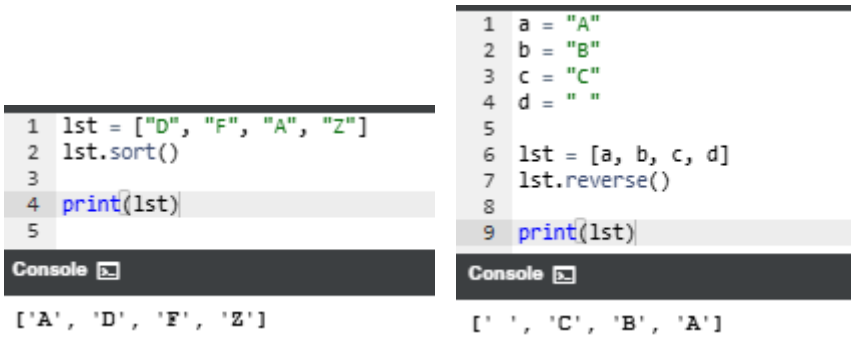


Рисунок 4.12 – Сортування списків з символами

Як бачимо з рисунку 4.12, сортування букв або слів відбувається за алфавітом, причому пробіл вважається «меншим» за букву (ставиться на початок списку при сортуванні).

Можна також створювати списки на основі змінних та сортувати їх (рисунок 4.13).

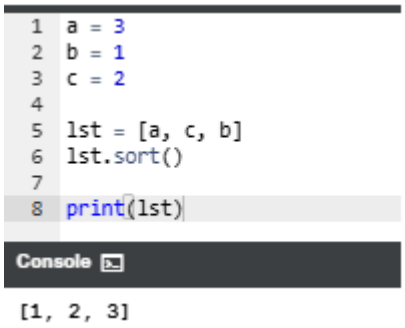


Рисунок 4.13 – Сортування списку зі змінних

Існує також метод списку з назвою `reverse()`, який можна використовувати для зворотного розташування елементів списку, наприклад:

```
lst = [5, 3, 1, 2, 4]
print(lst)

lst.reverse()
print(lst)  # виведе: [4, 2, 1, 3, 5]
```

4.10 Особливості зберігання списків в пам'яті

4.10.1 Змінні, що вказують на одну й ту саму область в пам'яті

Розглянемо приклад, наведений на рисунку 4.14.

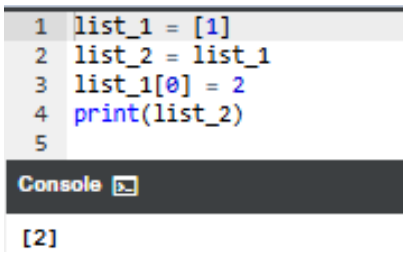


Рисунок 4.14 – Створення «копії» (псевдоніма) списку

Програма:

–створює одноелементний список з іменем `list_1`;

–присвоює його новому списку з іменем `list_2`;

–змінює єдиний елемент `list_1`;

–виводить `list_2`.

Несподіванкою є той факт, що програма виведе [2], а не [1], що здавалося очевидним результатом.

Списки (і багато інших складних структур Python) зберігаються інакше, ніж звичайні (скалярні) змінні.

Можна сказати і так:

–ім'я звичайної змінної – це ім'я її вмісту;

–ім'я списку – це ім'я комірки пам'яті, в якій зберігається список.

Присвоєння: `list_2 = list_1` копіює *назву масиву, а не його вміст*. По суті, дві назви (`list_1` та `list_2`) ідентифікують одне і те ж саме місце в пам'яті комп'ютера. Зміна однієї з них впливає на іншу, і навпаки. Можна провести аналогію – і за іменем, і за псевдонімом ми звертаємося до однієї й тої самої людини, так само сталося і в цьому прикладі (на рисунку 4.14) – `list_2` став псевдонімом для `list_1.`

4.10.2 Зрізи

Зріз – це елемент синтаксису Python, який дозволяє *зробити абсолютно нову копію списку або частини списку*.

Він дійсно копіює зміст списку, а не його назву.

Розглянемо приклад, наведений на рисунку 4.15. Як бачимо, він дуже подібний до прикладу з рисунку 4.14 (різниця лише в `[:]` вкінці другого рядка), проте результат різний.

```
1  list_1 = [1]
2  list_2 = list_1[:]
3  list_1[0] = 2
4  print(list_2)
5
Console
[1]
```

Рисунок 4.15 – Створення копії списку за допомогою зрізу

Ця непомітна частина коду, описана як `[:]`, здатна створювати абсолютно новий список.

Одна з найбільш загальних форм зрізу виглядає наступним чином:

```
my_list[start:end]
```

Як бачите, це нагадує індексацію, але двокрапка всередині має велике значення.

–`start` індекс першого елемента, що належить зрізу;

–`end` індекс першого елемента, який вже не входить до зрізу.

Зріз цієї форми *складає новий (цільовий) список, беручи елементи з вихідного списку – елементи індексів від start до end - 1*.

Примітка: не до `end`, а до `end - 1`. Елемент з індексом, що дорівнює `end`, є першим елементом, який *не бере участі у зрізі*.

Можливе використання від'ємних значень, як для `start`, так і для `end` (як і при індексації).

Розглянемо фрагмент коду:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[1:3]
print(new_list)
```

Список `new_list` матиме `end - start (3 - 1 = 2)` елементів, тобто, елементи з індексами `1` і `2` (але не `3`).

Результат роботи фрагменту має вигляд: `[8, 6].`

Приклади використання зрізів для копіювання списків наведено на рисунку 4.16.

```
1  # Копіювання всього списку.
2  list_1 = [1]
3  list_2 = list_1[:]
4  list_1[0] = 2
5  print(list_2)
6
7  # Копіювання частини списку.
8  my_list = [10, 8, 6, 4, 2]
9  new_list = my_list[1:3]
10 print(new_list)
11
Console
[1]
[8, 6]
```

Рисунок 4.16 – Створення копій списків за допомогою зрізів

Зі зрізами також можна використовувати від’ємні індекси (рисунок 4.17).

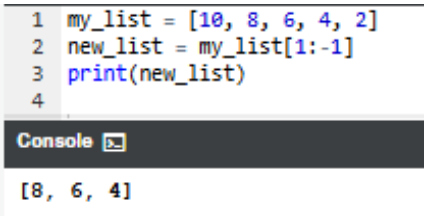


Рисунок 4.17 – Від’ємні індекси в зрізах

Якщо `start` вказує на елемент, що лежить далі, ніж той, що зазначений в `end` (від початку списку), то зріз буде *порожнім* (рисунок 4.18).

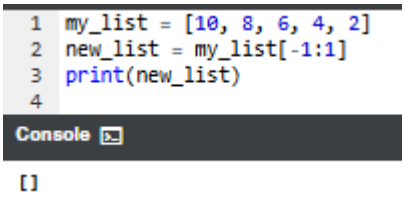


Рисунок 4.18 – Порожній результат зрізу

Якщо ви опускаєте `start` зрізу, то вважається, що ви хочете отримати зріз, який починається з елемента з індексом `0`.

Іншими словами, зріз такої форми:

```
my_list[:end]
```

є більш компактним аналогом:

```
my_list[0:end]
```

Подивіться на фрагмент нижче:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[:3]
print(new_list)
```

Його результат:

```
[10, 8, 6]
```

Аналогічно, якщо ви опускаєте `end` у вашому зрізі, то вважається, що ви хочете, щоб зріз закінчувався на елементі з індексом `len(my_list).`

Іншими словами, зріз такої форми:

```
my_list[start:]
```

є більш компактним еквівалентом:

```
my_list[start:len(my_list)]
```

Подивіться на наступний фрагмент:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[3:]
print(new_list)
```

Його результат:

```
[4, 2]
```

Як ми вже говорили раніше, якщо опустити `start` і `end`, то буде створено *копію всього списку*:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[:]
print(new_list)
```

Результат роботи фрагменту матиме вигляд:

```
[10, 8, 6, 4, 2]
```

Отже, якщо ви хочете скопіювати список або частину списку, ви можете зробити це, виконавши *зріз*:

```
colors = [‘червоний’, ‘зелений’, ‘помаранчевий’]

copy_whole_colors = colors[:]  # скопіювати весь список
copy_part_colors = colors[0:2]  # скопіювати весь список
```

4.10.3 Команда del

Описана раніше команда `del` здатна *видаляти не тільки один елемент списку за раз – вона може видаляти зрізи*:

```
my_list = [10, 8, 6, 4, 2]
del my_list[1:3]
print(my_list)
```

Примітка: у цьому випадку зріз *не створює жодного нового списку!*

Результат роботи фрагменту має вигляд:

```
[10, 4, 2].
```

Також можливе видалення *всіх елементів* одночасно:

```
my_list = [10, 8, 6, 4, 2]
del my_list[:]
print(my_list)
```

Список стає порожнім, і виводиться:

```
[].
```

Видалення зрізу з коду кардинально змінює його смисл.

Подивіться:

```
my_list = [10, 8, 6, 4, 2]
del my_list
print(my_list)
```

Інструкція `del` *видалить сам список, а не його вміст*.

Виклик функції `print()` з останнього рядка коду призведе до помилки під час виконання.

Тобто зріз (коли в коді є `[:]`) видаляє вміст списку, а його відсутність знищує сам список, а не його вміст.

4.11 Оператори in та not in

Python пропонує два дуже потужні оператори, здатні *переглядати список для того, щоб перевірити, чи зберігається певне значення всередині списку чи ні*.

Ці оператори:

```
elem in my_list
elem not in my_list
```

Перший з них (`in`) перевіряє, чи заданий елемент (лівий аргумент) в даний момент зберігається всередині списку (правий аргумент) – при виконанні цієї умови оператор повертає значення `True`.

Другий (`not in`) перевіряє, чи заданий елемент (лівий аргумент) відсутній у списку – при виконанні цієї умови оператор повертає значення `True`.

Приклади використання цих операторв наведено на рисунку 4.19.

```
1  my_list = [0, 3, 12, 8, 2]
2
3  print(5 in my_list)
4  print(5 not in my_list)
5  print(12 in my_list)
6
7
```

Console

```
False
True
True
```

Рисунок 4.19 – Приклад використання операторів `in` та `not in`

Задача 2. Знайти найбільший елемент в списку.

Розглянемо приклад, наведений на рисунку 4.20.

```
1  my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
2  largest = my_list[0]
3
4  for i in range(1, len(my_list)):
5      if my_list[i] > largest:
6          largest = my_list[i]
7
8  print(largest)
9
```

Console

```
17
```

Рисунок 4.20 – Приклад знаходження найбільшого елемента списку. Спосіб I

Ідея досить проста – ми тимчасово припускаємо, що перший елемент є найбільшим, і перевіряємо цю гіпотезу на решті елементів списку.

Використаємо іншу форму циклу `for` (рисунок 4.21).

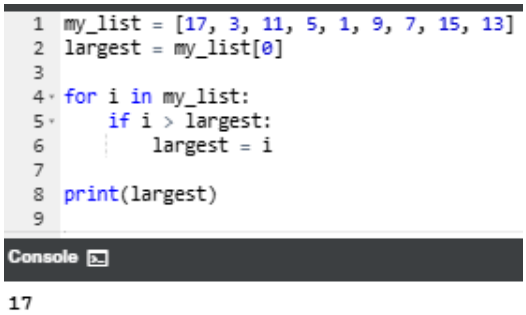


Рисунок 4.21 – Приклад знаходження найбільшого елемента списку. Спосіб II

У програмі з рисунку 4.21 виконується одне зайве порівняння, коли перший елемент порівнюється сам з собою, але це не є проблемою.

Якщо потрібно заощадити електроенергію комп'ютера, можна скористатися зрізом (рисунок 4.22).

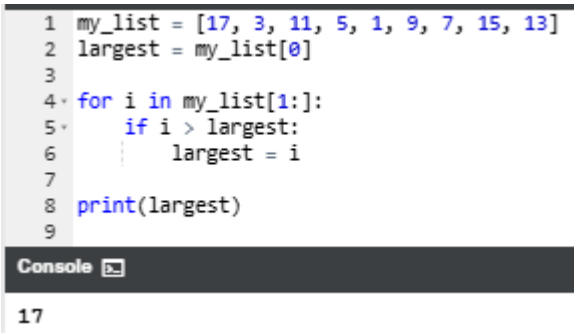


Рисунок 4.22 – Приклад знаходження найбільшого елемента списку. Спосіб III

Задача 3. Знайти місце розташування елемента 5 в списку.

Розв'язання цієї задачі наведено на рисунку 4.23.

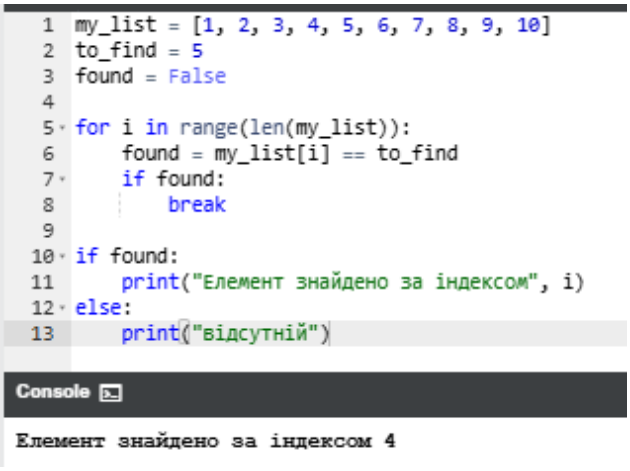


Рисунок 4.23 – Приклад знаходження елемента в списку

Примітка:

–кінцеве значення зберігається у змінній `to_find`;

–поточний стан пошуку зберігається у змінній `found (True/False);`

–коли `found` стає рівним `True`, відбувається вихід з циклу `for`.

Задача 4. Ви вибрали наступні номери в лотереї: `3, 7, 11, 42, 34, 49`. А випали такі числа: `5, 11, 9, 42, 3, 49`. Питання: скільки цифр ви вгадали?

Розв'язання цієї задачі наведено на рисунку 4.24.

```
1  drawn = [5, 11, 9, 42, 3, 49] # всі виграшні числа
2  bets = [3, 7, 11, 42, 34, 49] # ваші поточні ставки
3  hits = 0 #  ваші влучання
4
5  for number in bets:
6      if number in drawn:
7          hits += 1
8
9  print(hits)
```

Console

```
4
```

Рисунок 4.24 – Приклад розв'язання задачі про лотерею

4.12 Генерація псевдовипадкових чисел для заповнення списків

Програмний генератор випадкових чисел являє собою програму, яка генерує послідовність чисел за деяким алгоритмом. Завдяки алгоритму така послідовність чисел цілком детермінована, тобто не може бути цілком випадковою. Її називають послідовністю псевдовипадкових чисел.

В Python є вбудований модуль `random`, який дозволяє генерувати псевдовипадкові числа. Підключити модуль можна за допомогою інструкції `import`:

```
import random
```

Модуль random включає в себе функцію `random`, яка повертає дійсне число буде в діапазоні від `0.0` до `1.0`. Після імпортування модуля його назва стає змінною, через яку можна отримати доступ до атрибутів модуля:

`random.random()`

`random.randint(a, b)` – повертає випадкове ціле число на відрізку від `a` до `b` включно.

Приклади створення списків, заповнених випадковими числами, наведено на рисунку 4.25.

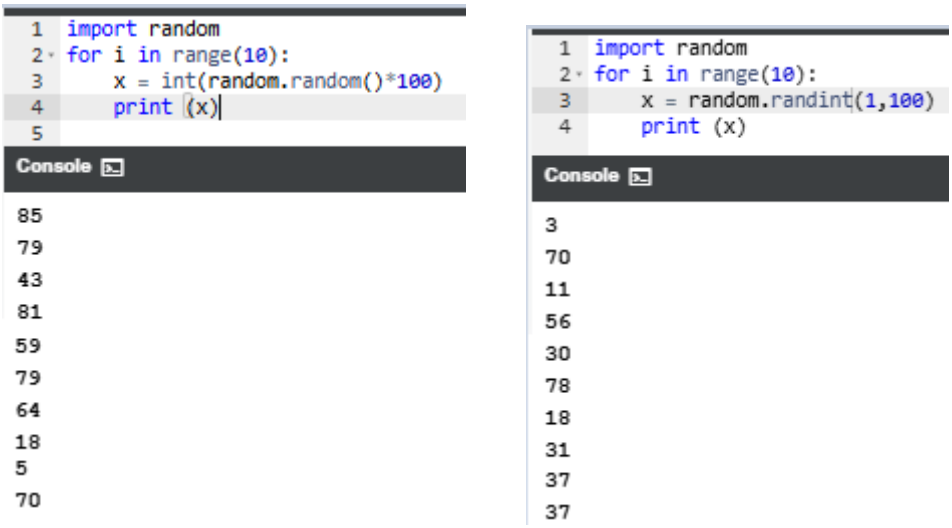


Рисунок 4.25 – Приклади заповнення списків випадковими величинами

Контрольні питання

1. Що таке список (`list`) у Python і для чого він використовується?
2. Як створити список у Python? Наведіть приклади різних способів створення списків.
3. Що таке індексація елементів у списку? З якого числа починається індексація?
4. Як отримати доступ до окремого елемента списку за його індексом?
5. Що означає негативна (від'ємна) індексація у списках? Наведіть приклад.
6. Як змінити значення елемента списку за індексом?
7. Як видалити елемент зі списку? Опишіть різні способи.
8. Як дізнатися довжину списку?
9. Які основні методи списків ви знаєте?
10. Як можна відсортувати список у Python?
11. У чому різниця між методами `sort()` і функцією `sorted()`?
12. Що відбувається при копіюванні списків через оператор `=`? Як уникнути спільного посилання на один об'єкт у пам'яті?
13. Як перевірити, чи певний елемент належить списку?
14. Як можна об'єднати два або більше списків?
15. Як створити зріз (slice) списку і що він повертає?

ТЕМА 5. Двовимірні масиви

5.1 Списки в списках, розуміння списків

Списки можуть складатися як зі скалярів (тобто чисел), так і з елементів набагато складнішої структури (такі приклади, як рядки, булеві чи навіть інші списки). Розглянемо докладніше випадок, коли елементами списку є самі списки.

Такі *масиви* дуже часто зустрічаються в нашому житті. Як двовимірний масив можна уявити стільці в залі кінотеатру, де кожний стілець визначається двома індексами – номером ряду і номером стільця в ряду. Ще одним прикладом може слугувати шахова дошка (рисунок 5.1).

Рисунок 5.1 – Шахова дошка

Шахова дошка складається з рядків і стовпчиків. В ній вісім рядків і вісім стовпчиків. Кожен стовпчик позначено літерами від `A` до `H`. Кожен рядок позначено цифрами від одного до восьми.

Розташування кожного поля ідентифікується парою «літера-цифра». Таким чином, ми знаємо, що лівий нижній кут дошки (в якому стоїть біла тура) – `A1`, а протилежний кут – `H8`.

Припустимо, що за допомогою пар «літера-цифра» ми можемо вказати будь-яку клітинку шахової дошки. Також можна припустити, що кожен рядок на шаховій дошці – це список.

Розглянемо код:

```
row = []

for i in range(8):
    row.append(WHITE_PAWN)
```

Він створює список з восьми елементів, що відповідають другому ряду шахової дошки – заповненому пішаками (припустимо, що `WHITE_PAWN` заздалегідь визначений символ, що позначає білого пішака).

Такого ж ефекту можна досягти за допомогою розуміння списків – спеціального синтаксису, який використовується в Python для заповнення великих списків.

Розуміння списку – створює той самий список, але на льоту, тобто під час виконання програми, а не описує його статично.

Погляньте на фрагмент:

```
row = [WHITE_PAWN for i in range(8)]
```

Частина коду, розміщена в дужках, конкретизує:

– дані, які будуть використані для заповнення списку (`WHITE_PAWN`);

– вираз, що описує, скільки разів дані зустрічаються всередині списку (`for i in range(8)`).

Розуміння списків забезпечує стислий спосіб створення списків. Загальні застосування полягають у створенні нових списків, де кожен елемент є результатом деяких операцій, застосованих до кожного члена іншої послідовності або ітерації, або створення підпослідовності тих елементів, які задовольняють певну умову.

Наприклад, припустимо, що ми хочемо створити список квадратів, наприклад:

```
squares = []
for x in range(10):
... squares.append(x**2)
print(squares)
```

Отримаємо результат:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Цей код створює (або перезаписує) змінну з назвою `x`, яка все ще існує після завершення циклу. Ми можемо обчислити список квадратів без побічних ефектів за допомогою:

```
squares = [x**2 for x in range(10)]
print(squares)
```

Останній код більш стислий і читабельний. Він також створює десятиелементний список, заповнений квадратами десяти цілих чисел, починаючи з нуля (`0, 1, 4, 9, 16, 25, 36, 49, 64, 81`).

Розуміння списку складається з дужок, які містять вираз, після якого йде речення `for`, потім нуль або більше речень `for` або `if`. Результатом буде новий список, отриманий в результаті обчислення виразу в контексті речень `for` і `if`, які слідують за ним.

Розглянемо деякі інші приклади розуміння списку.

```
twos = [2 ** i for i in range(8)]
```

Фрагмент створює восьмиелементний масив, що містить перші вісім степенів двійки (`1, 2, 4, 8, 16, 32, 64, 128`).

```
odds = [x for x in squares if x % 2 != 0 ]
```

Фрагмент формує список, який містить лише непарні елементи списку `squares`, який був створений раніше.

Отже, *розуміння списків* дозволяє створювати нові списки з існуючих в лаконічний і елегантний спосіб. Синтаксис розуміння списку виглядає наступним чином:

```
[expression for element in list if conditional]
```

що фактично є еквівалентом наступного коду:

```
for element in list:
    if conditional:
        expression
```

5.2 Двовимірні масиви

Також припустимо, що заданий елемент з ім'ям EMPTY позначає порожнє поле на шаховій дошці.

Отже, якщо ми хочемо створити список списків, який би відбивав усю шахову дошку, то для цього треба зробити наступне:

```
board = []

for i in range(8):
    row = [EMPTY for i in range(8)]
    board.append(row)
```

Примітка:

– внутрішня частина циклу формує рядок, що складається з восьми елементів (кожен з яких дорівнює EMPTY) і додає його до списку board;

– зовнішня частина повторює його вісім разів;

– всього список board складається з 64 елементів (всі дорівнюють EMPTY).

Ця модель ідеально імітує справжню шахову дошку, яка фактично виглядає як список з восьми елементів, розташованих в один ряд. Підсумуємо наші спостереження:

– елементами рядків є поля, по вісім у кожному рядку;

– елементами шахової дошки є рядки, по вісім на шахову дошку.

Змінна board тепер є *двовимірним масивом*. Його ще називають, за аналогією з алгебраїчним терміном, матрицею.

Оскільки списки можуть бути вкладеними, ми можемо скоротити створення дошки таким чином:

```
board = [[EMPTY for i in range(8)] for j in range(8)]
```

Внутрішня частина створює рядок, а зовнішня – список рядків.

Доступ до обраного поля дошки здійснюється за допомогою двох індексів – перший вибирає рядок, другий – номер поля в рядку, який де-факто є номером стовпчика.

Погляньте на шахову дошку. Кожне поле містить пару індексів, за якими можна отримати доступ до даних поля (рисунок 5.2).

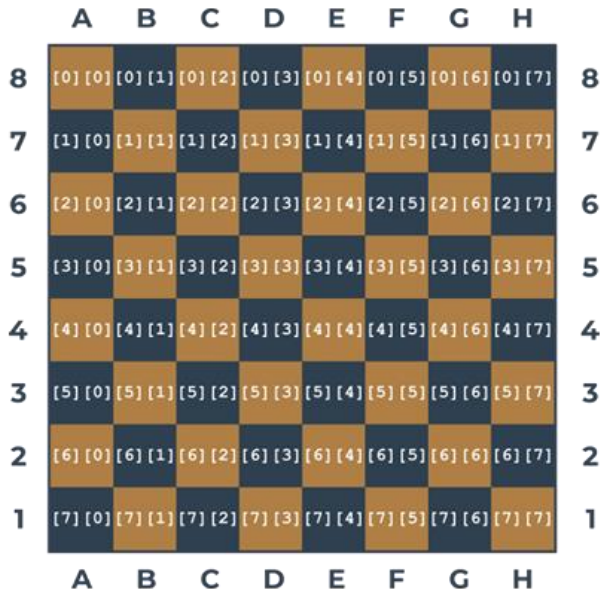


Рисунок 5.2 – Шахова дошка. Адресація комірок

Дивлячись на малюнок, зображений вище, давайте розставимо на дошці шахові фігури. Спочатку додамо всі тури:

```
board[0][0] = ROOK
board[0][7] = ROOK
board[7][0] = ROOK
board[7][7] = ROOK
```

Якщо ви хочете поставити коня на `C4`, то це робиться наступним чином:

```
board[4][2] = KNIGHT
```

А тепер пішак на `E5`:

```
board[3][4] = PAWN
```

Для створення *матриць* (тобто двовимірних списків) у мові Python можна застосовувати *вкладені списки*. Наприклад, для створення списку, зображеного на рисунку 5.3, можна скористатися кодом, що наведений нижче:

```
# Таблиця з 4-стовпчиків/4-рядків - двовимірний масив (4x4)
table = [[":(", ":)", ":(", ":)"],
```

```
        [":)", ":(", ":)", ":)"],
        [":(", ":)", ":)", ":("],
        [":)", ":)", ":)", ":("]]

print(table)
print(table[0][0])  # виведе: ‘:(‘
print(table[0][3])  # виведе: ‘:)’
```

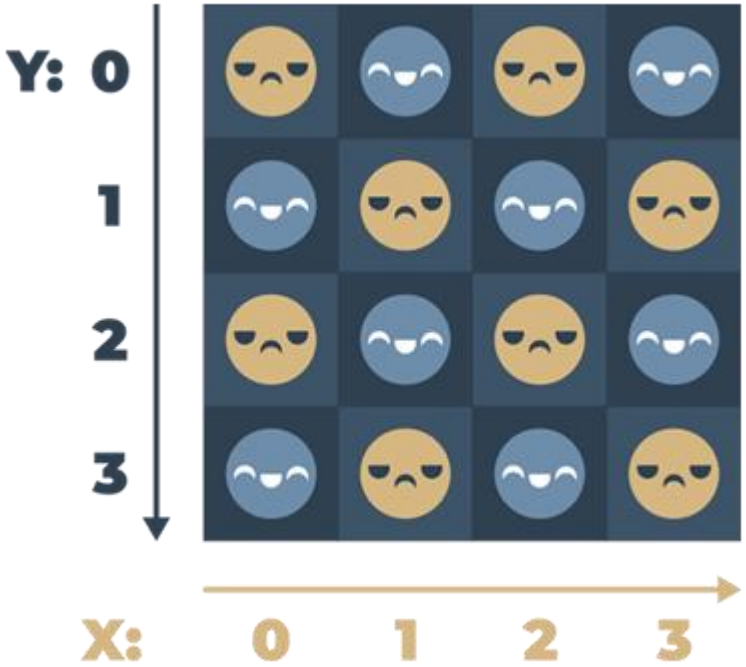


Рисунок 5.3 – Двовимірний масив емоджі

5.3 Багатовимірна природа списків

Давайте заглибимося в багатовимірну природу списків. Щоб знайти будь-який елемент двовимірного списку, потрібно використовувати дві координати:

– вертикальну (номер рядка);

– горизонтальну (номер стовпця).

Уявіть, що ви розробляєте частину програмного забезпечення для автоматичної метеостанції. Прилад фіксує температуру повітря щогодини і робить це протягом місяця. Це дає вам `24 × 31 = 744` записи. Спробуємо створити список, здатний зберігати всі ці результати.

По-перше, ви повинні вирішити, який тип даних буде відповідним для цього додатку. В даному випадку найкраще підійде `float`, оскільки термометр здатен вимірювати температуру з точністю до `0,1 ℃.`

Далі ви самостійно вирішуєте, що показання в рядках будуть записуватися кожну годину (таким чином, в рядку буде `24` елементи) і кожен з рядків буде відповідати певному дню місяця (припустимо, що в кожному місяці `31` день, отже, вам знадобиться `31` рядок). Наведемо відповідну пару позначень (`h` означає годину, `d` означає день):

```
temps = [[0.0 for h in range(24)] for d in range(31)]
```

Зараз вся матриця заповнена нулями. Можна припустити, що вона оновлюється автоматично за допомогою спеціальних апаратних агентів – сенсорів. Залишається лише дочекатися заповнення матриці даними вимірювань.

Тепер настав час визначити середньомісячну температуру опівдні. Підсумуйте всі `31` показники, зафіксовані опівдні, і розділіть суму на `31`. Можна припустити, що спочатку зберігається температура опівночі. Ось відповідний код:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# Матриця тут чарівним чином оновлюється.
#

total = 0.0

for day in temps:
    total += day[11]

average = total / 31

print("Середня температура опівдні:", average)
```

Примітка: змінна `day`, що використовується в циклі `for`, не є скаляром – кожен прохід по матриці `temps` присвоює їй наступні рядки матриці, отже, вона є списком. Вона повинна мати індекс `11`, щоб отримати доступ до значення температури, виміряної опівдні.

Тепер знаходимо найвищу температуру за весь місяць – дивіться код:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# Матриця тут чарівним чином оновлюється.
#

highest = -100.0

for day in temps:
    for temp in day:
        if temp > highest:
            highest = temp

print("Найвища температура була:", highest)
```

Примітка:

– змінна `day` перебирає всі рядки матриці `temps`;

– змінна `temp` перебирає всі вимірювання, зроблені протягом одного дня.

А тепер порахуйте дні, коли температура опівдні була не менше 20 °C:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# Матриця тут чарівним чином оновлюється.
#

hot_days = 0

for day in temps:
    if day[11] > 20.0:
        hot_days += 1

print(hot_days, "дні(в) була спека.")
```

Python не обмежує глибину включення списку в список. Нижче наведено приклад *тривимірного масиву*:

```
rooms = [[[False for r in range(20)] for f in range(15)] for t
in range(3)]
```

Уявіть собі готель. Це величезний готель, що складається з трьох корпусів, по 15 поверхів кожен. На кожному поверсі розташовано по 20 номерів. Для цього

потрібен масив, який може накопичувати та обробляти інформацію про зайняті/вільні номери.

Перший крок – тип елементів масиву. У цьому випадку підійде логічне значення (`True/False`).

Крок другий – спокійний аналіз ситуації. Підсумуємо наявну інформацію: три корпуси, 15 поверхів, 20 номерів.

Тепер ви можете створити масив:

```
rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]
```

Перший індекс (від 0 до 2) вказує на один з корпусів, другий (від 0 до 14) – на поверх, третій (від 0 до 19) – на номер кімнати. Всі кімнати спочатку вільні.

Наразі ви можете забронювати номер для двох молодят: другий корпус, десятий поверх, кімната 14:

```
rooms[1][9][13] = True
```

та звільнити другу кімнату на п’ятому поверсі першого корпусу:

```
rooms[0][4][1] = False
```

Перевірте, чи є вільні місця на 15-му поверсі третього корпусу:

```
vacancy = 0

for room_number in range(20):
    if not rooms[2][14][room_number]:
        vacancy += 1
```

Змінна `vacancy` дорівнює `0`, якщо всі номери зайняті, або перевірте, чи є вільні місця на 15-му поверсі третього корпусу:

```
vacancy = 0

for room_number in range(20):
    if not rooms[2][14][room_number]:
        vacancy += 1
```

в іншому випадку дорівнює кількості вільних номерів.

Ви можете вкладати скільки завгодно списків у списки, створюючи таким чином n-вимірні списки, наприклад, три-, чотири- або навіть шістдесят чотиривимірні масиви. Наприклад, з тривимірним масивом, наведеним на рисунку 5.4, можна працювати так:

```
# Куб - тривимірний масив (3x3x3)

cube = [[[':(', 'x', 'x'],
         [':)', 'x', 'x'],
         [':(', 'x', 'x']],

        [[':)', 'x', 'x'],
         [':(', 'x', 'x'],
         [':)', 'x', 'x']],

        [[':(', 'x', 'x'],
         [':)', 'x', 'x'],
         [':)', 'x', 'x']]]

print(cube)
print(cube[0][0][0])  # виведе: ':('
print(cube[2][2][0])  # виведе: ':)'
```

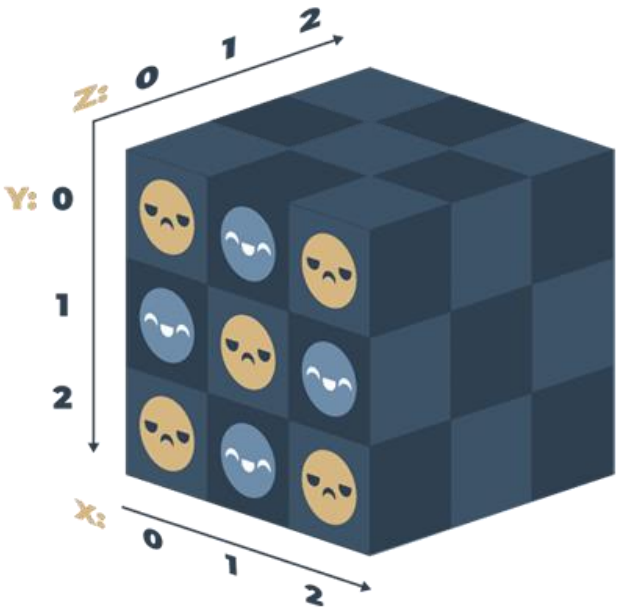


Рисунок 5.4 – Тривимірний масив

Збережемо дані таблиці, поданої на рисунку 5.5, у змінній a.

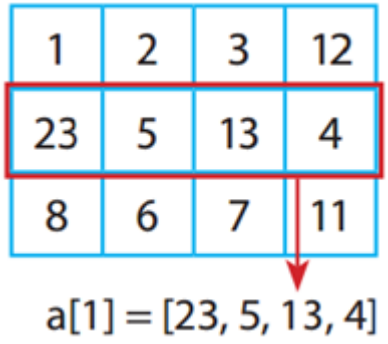


Рисунок 5.5 – Двовимірний масив

```
a = [[1, 2, 3, 12], [23, 45, 13, 4], [8, 6, 7, 11]]
```

Кожний елемент двовимірного списку `a` також є списком, що містить дані з одного рядка таблиці. Довжина списку `len(a) = 3`.

Звернутися до елемента двовимірного списку (рядка таблиці) можна за його індексом:

`a[0] = [1, 2, 3, 12].`

Для перебору рядків списку використовується цикл `for`.

Можна перебрати індекси вкладених списків – елементів двовимірного списку.

```
for i in range (len(a)):
        print(a[i])
```

Можна перебрати всі наявні у двовимірному списку елементи – рядки таблиці.

```
 for row in a:
        print(row)
```

При використанні обох варіантів циклу в консоль буде виведено весь двовимірний масив.

Положення елемента в таблиці визначається 2 індексами – номером рядка і номером стовпця, отже для доступу до елемента у вкладеному списку слід вказати два індекси (рисунок 5.6).

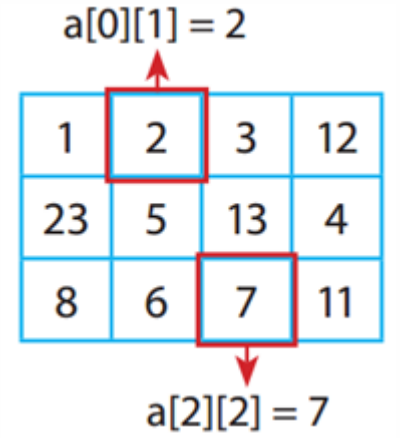


Рисунок 5.6 – Індексація елементів двовимірного масиву

Елемент, розташований на перетині `i`-го рядка і `j`-го стовпця масиву `a`, позначають `a[i][j]`. У двовимірних масивах перший індекс завжди вказує на номер рядка, а другий – на номер стовпця таблиці.

Виведення значень списку a в консоль наведено на рисунку 5.7.

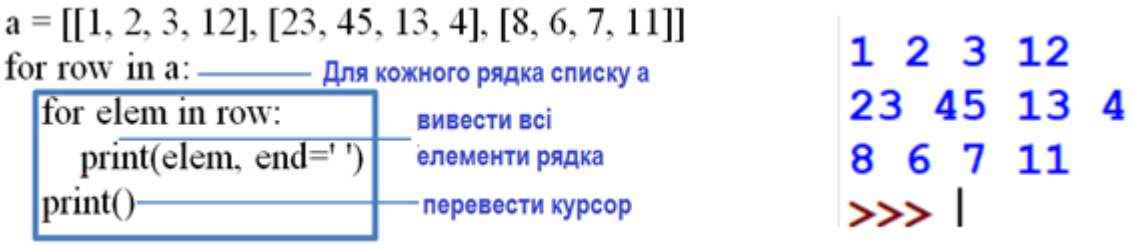


Рисунок 5.7 – Вивід елементів двовимірного масиву

Згадаємо, що параметр `end=‘ ‘` метода `print()` потрібен, щоб після виведення курсор залишився в тому самому рядку. Оператор `print()` з порожнім списком виведення переводить курсор на наступний рядок.

5.4 Створення та заповнення двовимірних списків

Створення порожнього списку `a` з `3` рядків:

```
a = [[], [], []]
```

Нехай потрібно створити список для збереження даних прямокутної таблиці, у якій n рядків і m стовпців, і заповнити його нулями.

Це можна зробити в такий спосіб:

```
n, m = 3, 4
a = [] # Створюємо порожній список
for i in range(n): # Додаємо n елементів
     a.append([0]*m) # Кожен елемент – список з m нулів
```

Отримаємо результат, як на рисунку 5.8.

```
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
```

Рисунок 5.8 – Двовимірний масив, заповнений нулями

Двовимірний масив можна створити з використанням генератора. Список `[0]*m` заново генерується для заповнення чергового елемента списку `a`.

```
a = [list(input().split()) for i in range(n)]
```

5.4.1 Уведення значень двовимірного масиву з клавіатури

Щоб заповнити двовимірний список цілими значеннями елементів двовимірного масиву з `n` рядків і `m` стовпців, уводячи значення елементів з клавіатури, слід `n` разів виконати дії:

1) увести рядок `s`, що містить `m` чисел, відокремлених пробілами, і розбити рядок s функцією `split()` на підрядки;
2) функцією `map()` кожний підрядок рядка s перетворити на числовий тип;
3) функцією `list()` отриману послідовність перетворити у список `row`;
4) список `row` додати до списку `a`.

```
n = int(input()) # Кількість рядків масиву
a = [] # Створюється порожній список
for i in range(n):
    s= input().split()
    row=list(map(int, s))
    a.append(row)
```

5.4.2 Заповнення двовимірного масиву випадковими числами

Заповнимо двовимірний список випадковими значеннями елементів двовимірного масиву з `n` рядків і `m` стовпців:

```
from random import randint
n, m=3, 4
a = [[] for i in range(n)]
for i in range(n):
            for j in range(m):
                        a[i] .append(randint(1,10))
# Додавання чергового елемента до i-го рядка
print (a)
```

Результат виведення списку a в консоль може бути таким:

```
[[7, 4, 4, 2], [10, 9, 1, 10], [6, 7, 2, 2]]
```

5.4.3 Надання значень елементам двовимірного масиву за формулою

Задання значень елементів двовимірного масиву за певною формулою має місце у випадках, коли значення елемента залежить від його індексів.

Приклад:

Заповнимо вкладений список a елементами таблиці Піфагора (рисунок 5.9).

```
n= m=9
a = [[] for i in range(n)]
for i in range(n):
    for j in range(m):
        a[i].append((i+1)*(j+1))
    print(a[i])
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 4, 6, 8, 10, 12, 14, 16, 18]
[3, 6, 9, 12, 15, 18, 21, 24, 27]
[4, 8, 12, 16, 20, 24, 28, 32, 36]
[5, 10, 15, 20, 25, 30, 35, 40, 45]
[6, 12, 18, 24, 30, 36, 42, 48, 54]
[7, 14, 21, 28, 35, 42, 49, 56, 63]
[8, 16, 24, 32, 40, 48, 56, 64, 72]
[9, 18, 27, 36, 45, 54, 63, 72, 81]
```

Рисунок 5.9 – Двовимірний масив, заповнений за формулою

Нехай кожний елемент масиву `5x5` дорівнює більшому з його індексів (рисунок 5.10).

```
a = [[], [], [], [],[] ]
for i in range(5):
    for j in range(5):
        a[i]=a[i]+[max(i,j)]
for i in range(5):
    for j in range(5):
        print(a[i][j], end=' ')
    print()
```

```
0 1 2 3 4
1 1 2 3 4
2 2 2 3 4
3 3 3 3 4
4 4 4 4 4
```

Рисунок 5.10 – Заповнення двовимірного масиву

Заповнимо двовимірний масив `5 × 5` у такий спосіб:

– елементам головної діагоналі (індекси рівні) присвоїти значення `1`;

– елементам, що розташовані вище головної діагоналі, – значення `2`;

– елементам, що розташовані нижче головної діагоналі, – значення `0` (рисунок 5.11).

–

```
a = [[], [], [], [],[] ]
for i in range(5):
    for j in range(5):
        if i == j: a[i].append(1)
        elif i<j: a[i].append(2)
        else: a[i].append(0)
```

```
1 2 2 2 2
0 1 2 2 2
0 0 1 2 2
0 0 0 1 2
0 0 0 0 1
```

Рисунок 5.11 – Заповнення двовимірного масиву

Контрольні питання

1. Що означає поняття «багатовимірний список» у Python?
2. Як створити список у списку (вкладений список)? Наведіть приклад.
3. Як у Python реалізується двовимірний масив за допомогою списків?
4. Як звернутися до окремого елемента двовимірного списку за індексами?
5. Як отримати рядок або стовпець із двовимірного списку?
6. Як змінити значення конкретного елемента в багатовимірному списку?
7. Як можна перебрати всі елементи двовимірного списку за допомогою вкладених циклів?
8. Як створити порожній двовимірний список і поступово додавати до нього елементи?
9. Як перевірити розмірність списку (кількість рядків і стовпців)?

10. Як можна використати списки в списках для зберігання таблиць або матриць даних?

11. Як ініціалізувати двовимірний список за допомогою генераторів списків?

12. Як відобразити вміст двовимірного списку у зручному табличному вигляді?

13. Які практичні приклади застосування багатовимірних списків можна навести?

ТЕМА 6. Функції користувача

6.1 Декомпозиція задачі

Якщо вам потрібно вивести якісь дані на консоль, то ви використовуєте функцію `print()`. Коли ви хочете отримати значення для змінної, ви використовуєте функцію `input()` у поєднанні з функціями `int()` або `float()`. Ви також використовували деякі методи, які фактично є функціями, але оголошуються дуже специфічним чином.

Буває, що певний фрагмент коду багато разів повторюється у програмі. Він повторюється або буквально, або з невеликими змінами, що полягають у використанні інших змінних у тому ж алгоритмі. Буває й таке, що програміст не може втриматися від спрощення своєї роботи, і починає клонувати такі шматки коду, використовуючи буфер обміну та операцію copy-paste.

Функція – це пойменований блок програмного коду, до якого можна звернутися з будь-якого місця програми необхідну кількість разів.

Якщо певний фрагмент коду починає з'являтися більш ніж в одному місці, розгляньте можливість його ізоляції у вигляді функції, яка буде викликатися з точок, де до цього був розміщений оригінальний код.

Може статися так, що алгоритм, який ви плануєте реалізувати, настільки складний, що код починає неконтрольовано розростатися, і раптом ви помічаєте, що вже не в змозі так легко в ньому орієнтуватися.

Можна спробувати впоратися з цією проблемою, детально коментуючи код, але занадто багато коментарів робить код більшим і складнішим для розуміння. Вважають, що добре написана функція має бути зрозумілою з першого погляду.

Хороший програміст ділить код (задачу) на чітко відокремлені частини, і кожну з них кодує у вигляді окремої функції (рисунок 6.1).

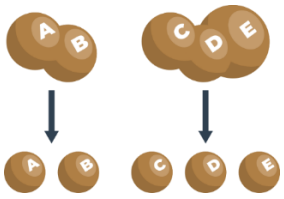


Рисунок 6.1 – Декомпозиція задачі

Це суттєво спрощує роботу програми, адже кожен фрагмент коду можна кодувати та тестувати окремо. Описаний процес часто називають *декомпозицією*.

Така декомпозиція продовжується до тих пір, поки не вийде набір коротких функцій, простих для розуміння і тестування.

Буває, що проблема настільки велика і складна, що її неможливо доручити одному розробнику, і над нею доводиться працювати команді програмістів. Проблему треба розподілити між кількома розробниками так, щоб забезпечити їх ефективну та злагоджену співпрацю. Цей вид декомпозиції має іншу мету, ніж описана раніше – мова не тільки про розподіл роботи, а й про розподіл відповідальності між розробниками. Кожен з них пише чітко визначений і розписаний набір функцій, які при об'єднанні в модуль дадуть кінцевий продукт.

Отже, функції варто створювати у трьох випадках:

– якщо певний фрагмент коду починає з'являтися більш ніж в одному місці, розгляньте можливість його ізоляції у вигляді функції, яка буде викликатися з точок, де до цього був розміщений оригінальний код;

– якщо фрагмент коду стає настільки великим, що його читання та розуміння може викликати проблему, розгляньте можливість розбиття його на окремі, менші задачі, і реалізуйте кожну з них у вигляді окремої функції;

– якщо ви збираєтеся розділити роботу між кількома програмістами, розкладіть проблему так, щоб продукт можна було реалізувати як набір окремо написаних функцій, зібраних разом в окремих модулях.

Функції в Python можуть з'являтися з кількох джерел:

– вбудовані функції (built-in functions) – це функції, які вже включені в сам Python. Наприклад:

```
print("Привіт")
len("рядок")
sum([1, 2, 3])
```

Приклади: print(), len(), input(), sum(), type(), range() і багато інших.

Повний перелік вбудованих функцій Python: https://docs.python.org/3/library/functions.html

– функції користувача (user-defined functions) – ми можемо самі створювати функції, використовуючи ключове слово `def`:

```
def hello(name):
    print(f"Привіт, {name}!")

hello("Ірина")
```

– функції з модулів (бібліотек) – Python має багато стандартних бібліотек (і ще більше сторонніх), з яких можна імпортувати функції:

```
import math
print(math.sqrt(16))  # функція sqrt з модуля math
```

Або:

```
from random import randint
print(randint(1, 10))
```

– анонімні (lambda) функції – це короткий спосіб створити функцію «на льоту»:

```
add = lambda x, y: x + y
print(add(3, 5))  # Виведе 8
```

– функції з фреймворків або сторонніх бібліотек – можна встановити бібліотеки через `pip`, наприклад `numpy`, `pandas`, `flask` тощо, а вони містять свої функції:

```
import numpy as np
np.mean([1, 2, 3])
```

Щоб дізнатися які функції є в бібліотеці, можна написати код, як у прикладі на рисунку 6.2.

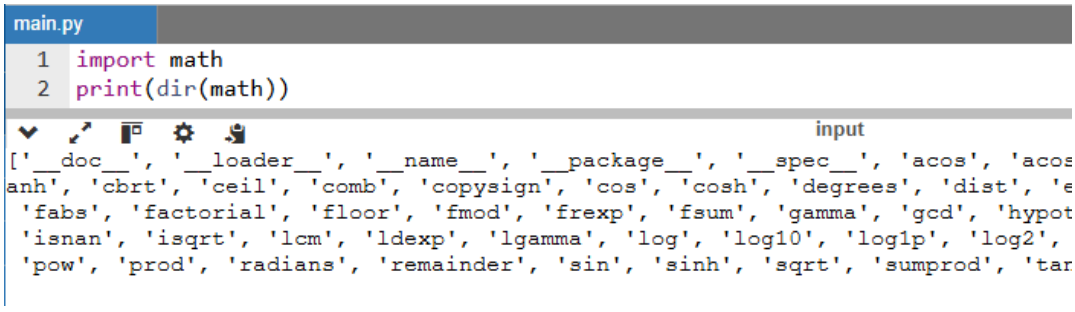


Рисунок 6.2 – Вивід переліку функцій в бібліотеці `math`

А щоб зрозуміти що робить конкретна функція, можна скористатися довідкою (рисунок 6.3).

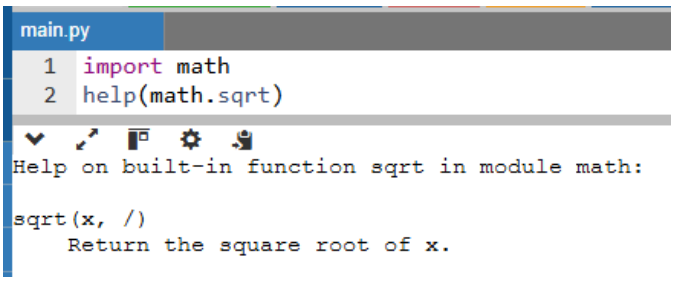


Рисунок 6.3 – Вивід довідки про функцію `sqrt` бібліотеки `math`

6.2 Створення власних функцій

Так виглядає найпростіше визначення функції:

```
def function_name():
    function_body
```

Функція завжди починається з ключового слова `def` (для оголошення).

Після `def` йде *ім'я функції* (правила іменування функцій такі самі, як і для назв змінних).

Після імені функції передбачена наявність пари *круглих дужок* (обов'язково). Рядок повинен закінчуватися *двокрапкою*.

У наступному після `def` рядку починається *тіло функції* – декілька (хоча б одна) обов'язково вкладених інструкцій, які будуть виконуватись при кожному виклику функції; зверніть увагу: функція закінчується там, де закінчується вкладеність, тому треба бути уважним.

Викликаємо (звертаємось до функції) за допомогою її імені, дужок та аргументів у цих дужках. Приклад наведено на рисунку 6.4.

При виклику функції Python запам'ятовує місце, де це відбулось, і переходить безпосередньо до функції, після чого виконується тіло функції. Досягнення кінця функції змушує Python повернутись на місце відразу після точки виклику функції.

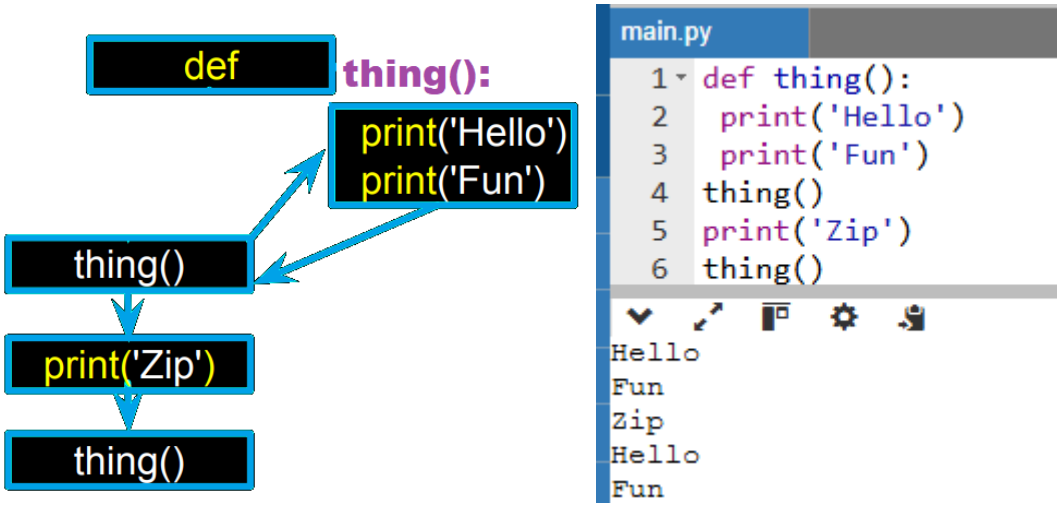


Рисунок 6.4 – Приклад створення та використання власної функції

Python виконує код згори донизу. Він не буде заглядати вперед, щоб знайти функцію, яку ми забули поставити в належне місце тобто функцію завжди треба описувати перед її викликом).

Не можна щоб в одній програмі були функція та змінна з однаковими іменами.

Наступний фрагмент призведе до помилки:

```
def message():
    print("Введіть значення: ")

message = 1
```

Присвоєння значення імені `message` призводить до того, що Python забуває його попередню роль. Функція з назвою `message` стає недоступною.

6.3 Параметризовані функції

Ви можете оголосити функцію, яка не отримує жодних аргументів, наприклад:

```
def message(): # визначення функції
    print("Привіт") # тіло функції

message() # виклик функції
```

Ви можете визначити функцію, яка приймає аргументи, так само, як і функція з одним параметром, наведена нижче:

```
def hello(name): # визначення функції
    print("Привіт,", name) # тіло функції


name = input("Введіть своє ім’я: ")

hello(name) # виклик функції
```

Повністю потенціал функції розкривається тоді, коли вона може бути доповнена інтерфейсом, здатним приймати дані, що надходять від користувача. Такі дані можуть модифікувати поведінку функції, роблячи її більш гнучкою та адаптивною до мінливих умов.

Параметр – це фактично змінна, але є два важливих фактори, які роблять їх іншими та унікальними:

– параметри існують тільки всередині функцій, в яких вони були визначені, і єдиним місцем, де параметр може бути визначений, буде простір між парою круглих дужок в інструкції `def`;

– присвоєння значення параметру здійснюється в момент виклику функції, за допомогою вказування відповідного аргументу.

```
def function(parameter):
    ###
```

Не забувайте:

– *параметри* існують всередині функцій (це їх природне середовище);

– *аргументи* існують за межами функцій і є носіями значень, що передаються відповідним параметрам.

Між цими двома світами існує чітка і однозначна межа.

Задання одного або декількох параметрів при оголошенні функції змушу нас використовувати таку саму кількість аргументів при виклику функції, інакше виникне помилка (рисунок 6.5).

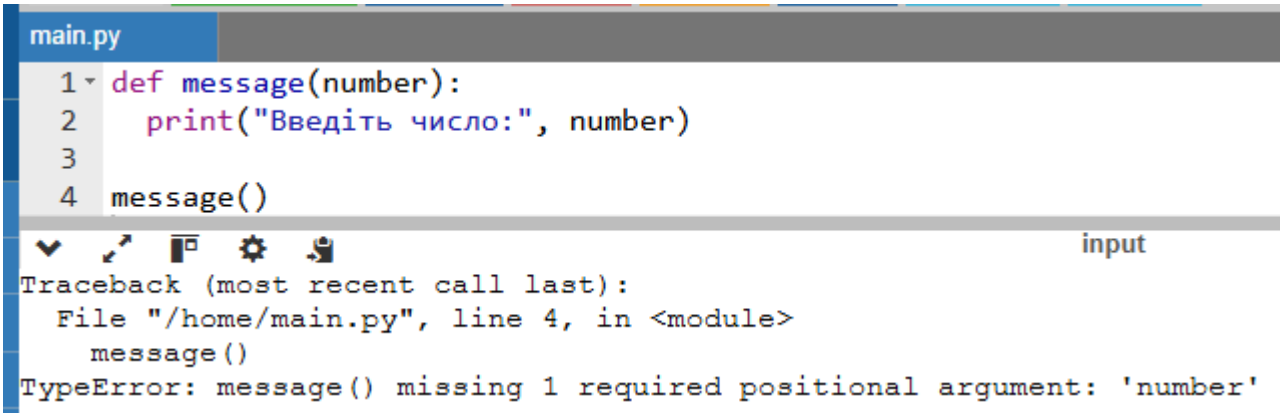


Рисунок 6.5 – Приклад неправильного виклику функції

Змінна за межами функції може мати ту саму назву, що і параметр функції (рисунок 6.6).

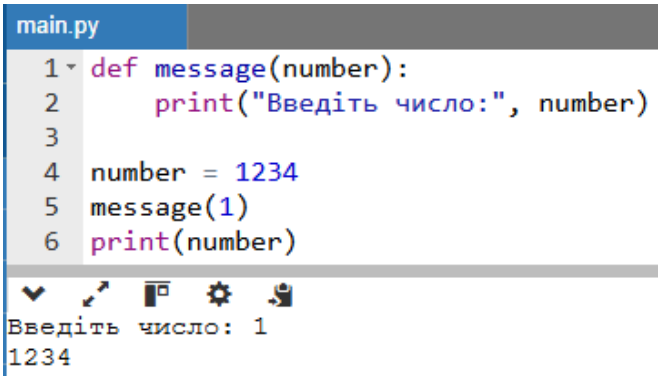


Рисунок 6.6 – Використання змінної та параметра `number`

Така ситуація запускає механізм, який називається затіненням: параметр `number` затінює будь-яку однойменну змінну, але тільки всередині функції, що визначає параметр.

Параметр з іменем `number` є зовсім іншою сутністю, ніж змінна з іменем `number`.

Функція може мати скільки завгодно параметрів, але чим більше у вас параметрів, тим важче запам’ятати їх ролі та призначення.

Давайте модифікуємо функцію – тепер вона має *два параметри*:

```
def message(what, number):
    print("Введіть", what, "число", number)
```

Це також означає, що виклик функції вимагатиме *двох аргументів* (рисунок 6.7).

main.py

```
1  def message(what, number):
2      print("Введіть", what, "число", number)
3
4  message("в телефоні", 11)
5  message("на клавіатурі", 5)
6  message("число", "число")
```

```
Введіть в телефоні число 11
Введіть на клавіатурі число 5
Введіть число число число
```

Рисунок 6.7 – Приклад функції з двома параметрами

Прийом, при якому `i`-му (першому, другому і т.д.) параметру функції присвоюється `i`-й (перший, другий і т.д.) аргумент, називається *позиційною передачею параметрів*, а аргументи, що передаються таким чином, називаються позиційними аргументами:

```
print(a, b, c)

my_function(1, 2, 3)
```

Примітка: передача позиційного параметра інтуїтивно використовується людьми в багатьох соціальних ситуаціях. Наприклад, може бути загальноприйнятим, що коли британці представляються, то називають своє ім'я перед прізвищем, наприклад, «Мене звати Джон Доу» (рисунок 6.8). Українці звикли це робити у зворотному порядку, наприклад, «Мене звати Лавренчук Світлана» (рисунок 6.9).

main.py

```
1  def introduction(first_name, last_name):
2      print("Привіт, мене звати", first_name, last_name)
3
4  introduction("Люк", "Скайвокер")
5  introduction("Джессі", "Уілсон")
6  introduction("Кларк", "Кент")
```

```
Привіт, мене звати Люк Скайвокер
П?віт, мене звати Джессі Уілсон
Привіт, мене звати Кларк Кент
```

Рисунок 6.8 – Британський звичай (позиція представлення)

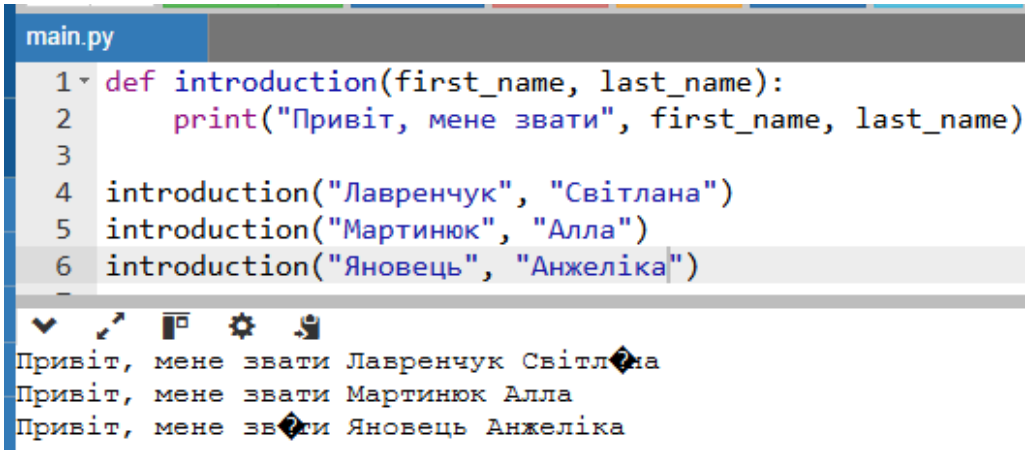


Рисунок 6.9 – Український звичай (позиція представлення)

Python пропонує іншу конструкцію передачі аргументів, коли *значення аргументу визначається його іменем*, а не позицією – це називається *передача аргументу за ключовим словом* (рисунок 6.10).

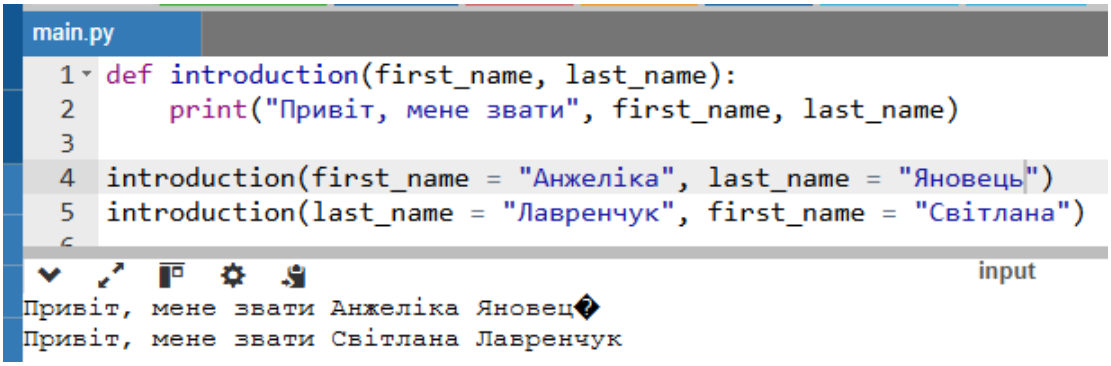


Рисунок 6.10 – Передача аргументу за ключовим словом

Ідея зрозуміла – перед значеннями, що передаються в параметрах, вказуються імена цільових параметрів, після яких ставиться знак `=`.

Позиція тут не має значення – значення кожного аргументу відоме на основі використаного імені.

6.4 Змішування позиційних і іменованих аргументів

Якщо хочете, то ви можете змішати обидва варіанти, але є одне непорушне *правило*: позиційні аргументи потрібно ставити перед аргументами ключових слів.

Щоб показати, як це працює, ми використаємо просту функцію з трьома параметрами:

```
def adding(a, b, c):
    print(a, "+", b, "+", c, "=", a + b + c)
```

Її призначення – оцінка та представлення суми всіх її аргументів.

Функція, викликана таким чином:

```
adding(1, 2, 3)
```

виведе:

```
1 + 2 + 3 = 6
```

Це був чистий приклад передачі позиційних аргументів.

Звичайно, можна замінити такий виклик варіантом з ключовим словом:

```
adding(c = 1, a = 2, b = 3)
```

Наша програма виведе такий рядок:

```
2 + 3 + 1 = 6
```

Зверніть увагу на порядок значень.

Спробуємо змішати обидва стилі.

Подивіться на виклик функції нижче:

```
adding(3, c = 1, b = 2)
```

Тут аргумент (3) для параметру `a` передається позиційним способом; аргументи для `c` та `b` вказані як ключові слова.

Результат:

```
3 + 2 + 1 = 6
```

Якщо ми спробуємо передати більше одного значення одному аргументу, отримаємо лише помилку виконання.

Подивіться на виклик нижче – таке враження, що ми намагалися задати `a` двічі:

```
adding(3, a = 1, b = 2)
```

Відповідь Python:

```
TypeError: adding() got multiple values for argument ’a’
```

Іноді трапляється так, що значення певного параметра використовуються частіше, ніж інші. Такі аргументи можуть мати *значення за замовчуванням* (наперед визначені), якщо їх відповідні аргументи були відсутні.

Кажуть, що одне з найпопулярніших українських прізвищ Шевченко. Спробуємо це врахувати.

Значення параметра за замовчуванням задається за допомогою зрозумілого та наочного синтаксису:

```
def introduction(first_name, last_name=" Шевченко "):
     print("Привіт, мене звати", first_name, last_name)
```

Достатньо лише доповнити ім’я параметра знаком =, за яким слідує значення за замовчуванням.

Викличемо функцію звичайним способом:

```
introduction("Світлана","Лавренчук")
```

Результат:

```
Привіт, мене звати Світлана Лавренчук
```

Як бачимо, нічого не змінилося, ось так:

```
introduction("Світлана")
```

або так:

```
introduction(first_name="Марія")
```

помилки не буде, і обидва виклики будуть успішними, а консоль виведе результат (рисунок 6.11).

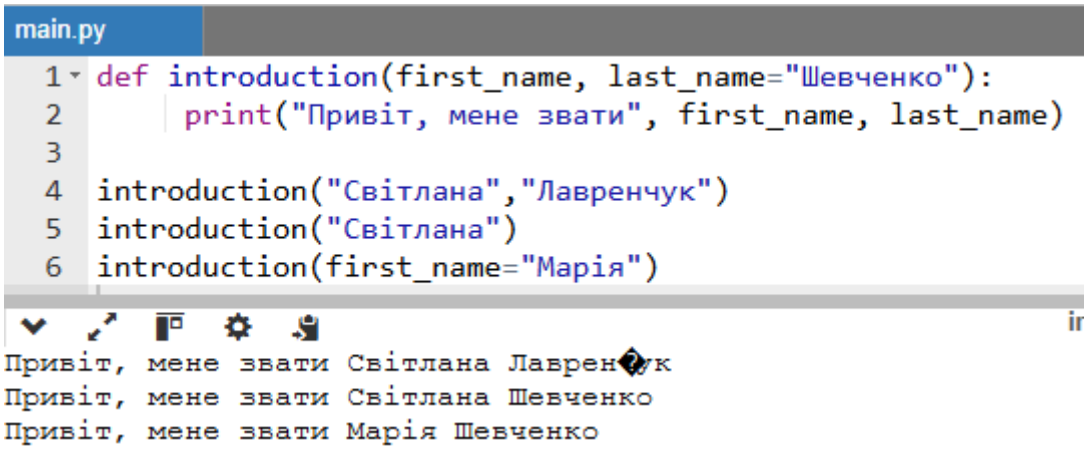


Рисунок 6.11 – Використання одного значення за замовчуванням

Обидва параметри можуть мати значення за замовчуванням (рисунок 6.12).

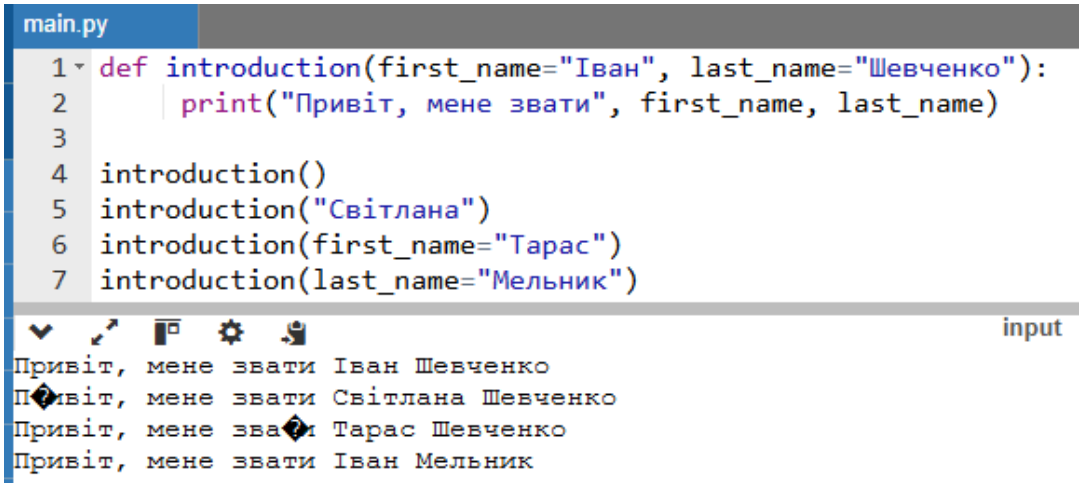


Рисунок 6.12 – Використання двох значень за замовчуванням

6.5 Повернення результату з функції

Всі наведені раніше функції мають певний ефект – вони створюють якийсь текст і спрямовують його на консоль.

Звичайно, функції, як і їхні математичні родичі, теж мають результати.

Щоб змусити *функції повернути значення* (але не тільки для цього) використовується інструкція `return` (це *ключове слово* Python).

Інструкція `return` має *два різних варіанти* – розглянемо їх окремо.

6.5.1 return без повернення результату

```
def happy_new_year(wishes = True):
```

```
    print("Три...")
    print("Два...")
    print("Один...")
    if not wishes:
        return

    print("Щасливого Нового року!")
```

Коли звертаємося без жодних аргументів:

```
happy_new_year()
```

функція створює невеликий шум – результат буде виглядати наступним чином:

```
Три...
Два...
Один...
Щасливого Нового року!
```

Якщо ми передамо в якості аргументу `False`:

```
happy_new_year(False)
```

то функція змінить поведінку функції – інструкція `return` призведе до її завершення безпосередньо перед виконанням умови – так виглядає оновлений результат:

```
Три...
Два...
один...
```

6.5.2 return з поверненням результату

Другий варіант `return` *розширений, з поверненням результату::*

```
def function():
    return expression
```

Є два результати від його використання:

– призводить до *негайного припинення виконання функції* (нічого нового порівняно з першим варіантом);

– крім того, функція обчислить значення виразу і поверне його (звідси і назва) в якості результату функції.

Розглянемо приклад:

```
def boring_function():
    return 123

x = boring_function()

print("Функція boring_function повернула свій результат. Він:", x)
```

Фрагмент виводить на консоль наступний текст:

```
Функція boring_function повернула свій результат. Він: 123
```

Розглянемо рисунок 6.13.

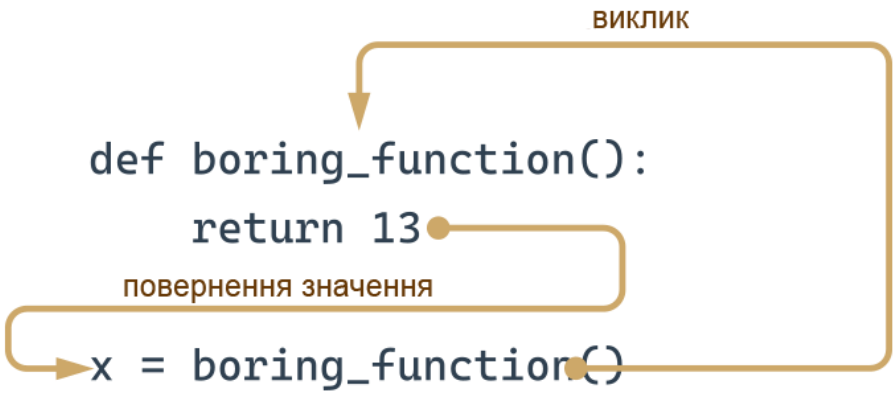


Рисунок 6.13 – Повернення значення функцією

Інструкція `return`, доповнена виразом (вираз тут дуже простий), «переносить» результат виразу в те місце, де була викликана функція. Результат може бути вільно використаний, наприклад, для присвоєння змінній. Він також може бути повністю проігнорований і безслідно загублений.

Не забувайте:

– можна *проігнорувати результат функції*, а задовольнитися лише ефектом функції (якщо він у неї є):

– якщо функція передбачає повернення результату, то вона повинна мати другий варіант інструкції `return`.

Існує ключове слово None, яке можна використовувати:

– для присвоєння його змінній (або повернення як результату функції);

– для порівняння його зі змінною для діагностики її внутрішнього стану.

Приклад:

```
value = None
if value is None:
    print("Вибачте, ви не маєте жодного значення")
```

Пам’ятайте: якщо функція не повертає певне значення за допомогою інструкції return, то вважається, що вона *неявно повертає* None.

Приклад:

```
def strange_function(n):
   if(n % 2 == 0):
       return True
```

Очевидно, що функція strange_function повертає значення True, якщо її аргумент парний, в іншому випадку вона поверне None (рисунок 6.14).

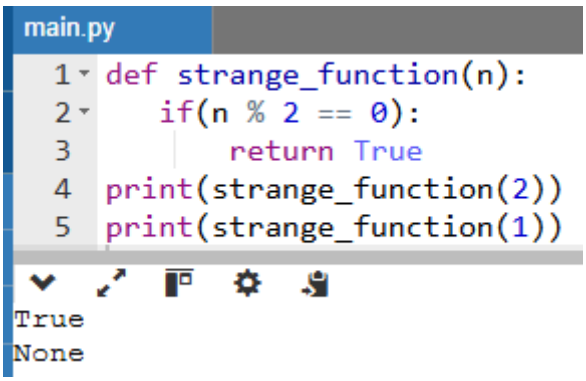


Рисунок 6.14 – Ключове слово None

У функцію можна передавати різні об’єкти, наприклад, списки (але і тіло функції повинно опрацьовувати ці об’єкти):

```
def list_sum(lst):
    s = 0

    for elem in lst:
        s += elem

    return s
```

```
print(list_sum([5, 4, 3]))
```

поверне в результаті 12.

Якщо викликати функцію так:

```
print(list_sum(5))
```

то виникне помилка, бо ми пробуємо передати замість списку звичайну цілу змінну:

```
TypeError: ‘int’ object is not iterable
```

Це пов’язано з тим, що цикл `for` не може повторюватися за одним цілим значенням.

Результатом функції може бути будь-який об’єкт, що розуміється мовою Python.

Приклад:

```
def strange_list_fun(n):
    strange_list = []

    for i in range(0, n):
       strange_list.insert(0, i)

    return strange_list
print(strange_list_fun(5))
```

Результат роботи програми буде виглядати наступним чином:

```
 [4, 3, 2, 1, 0]
```

6.6 Функції та області видимості

Область видимості імені (наприклад, імені змінної) – це частина коду, в якій ім’я може бути належним чином розпізнане.

Наприклад, областю видимості параметра функції є сама функція. Параметр недоступний за межами функції.

Приклад наведено на рисунку 6.15.

```
main.py
1 def scope_test():
2     x = 123
3
4 scope_test()
5 print(x)

Traceback (most recent call last):
  File "/home/main.py", line 6, in <module>
    print(x)
          ^
NameError: name 'x' is not defined
```

Рисунок 6.15 – Вихід за межі області видимості

Проте, змінна, що існує за межами функції, має область видимості всередині тіла функції (рисунок 6.16).

```
main.py
1 def my_function():
2     print("Чи знаю я цю змінну?", var)
3
4 var = 1
5 my_function()
6 print(var)

Чи знаю я цю змінну? 1
1
```

Рисунок 6.16 – Зовнішня змінна всередині функції

Розглянемо приклад, наведений на рисунку 6.17.

```
main.py
1 def my_function():
2     var = 2
3     print("Чи знаю я цю змінну?", var)
4
5 var = 1
6 my_function()
7 print(var)

Чи знаю я цю змінну? 2
1
```

Локальна змінна

Зовнішня змінна

Рисунок 6.17 – Локальні та зовнішні змінні

Змінна `var` , створена всередині функції, не є тією ж самою, що і змінна визначенна за її межами – існують дві різні змінні з однаковими іменами; при використанні всередині функції, локальна змінна затінює змінну, що надходить із зовнішнього світу.

Змінна, що оголошена за межами функції, має область видимості в середині тіла функції, за винятком таких, що оголошені в середині тіла функції з таким самим ім’ям.

Це також означає, що область видимості змінної, яка існує за межами функції, підтримується тільки при одержанні нею значення (читанні). Присвоєння значення призводить до створення власної змінної в функції.

В Python є спеціальний метод, який може розширити область видимості змінної таким чином, щоб вона входила до тіла функції (навіть якщо ви хочете не тільки прочитати значення, але й модифікувати його). Такий ефект спричиняє ключове слово `global`.

Використання цього ключового слова всередині функції з іменем (або іменами через кому) змінної (або змінних) змушує Python утриматися від створення нової змінної всередині функції – замість неї буде використана доступна ззовні.

Іншими словами, це ім’я стає глобальним (воно має *глобальну область видимості*, і неважливо, чи йдеться про читання, чи про присвоєння) (рисунок 6.18).

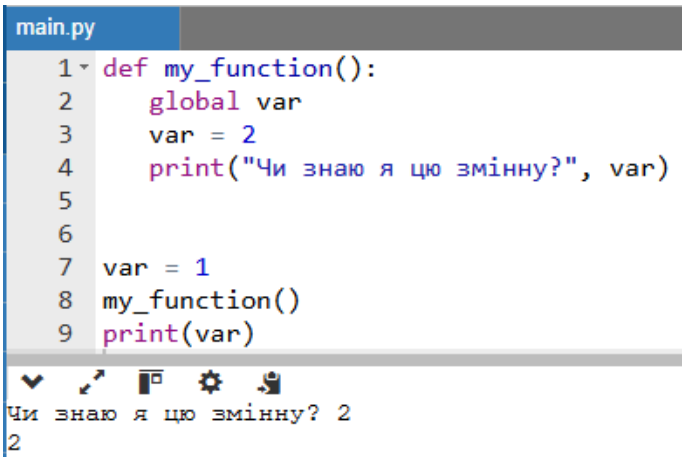


Рисунок 6.18 – Глобальна змінна

Не може бути в одній програмі двох глобальних змінних з однаковими іменами (рисунок 6.19).

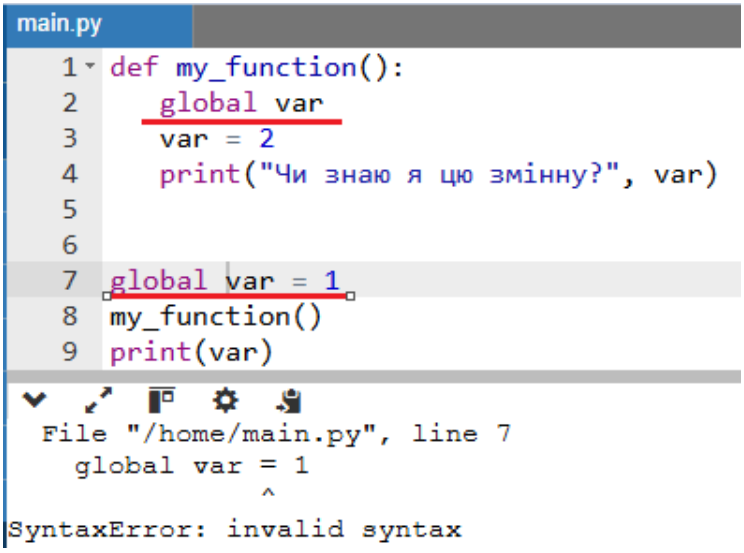


Рисунок 6.19 – Помилкове визначення глобальних змінних

Зміна значення параметра не поширюється за межі функції (рисунок 6.20).

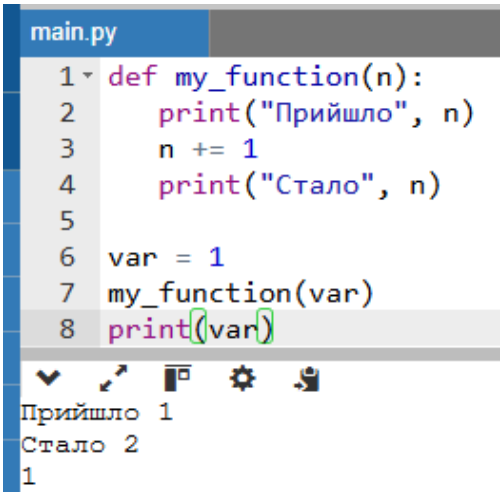


Рисунок 6.20 – Помилкове визначення глобальних змінних

Це також означає, що функція отримує *значення аргументу*, а не сам аргумент. Це вірно для скалярних величин. Розглянемо приклад передачі списків як параметрів функції (рисунок 6.21).

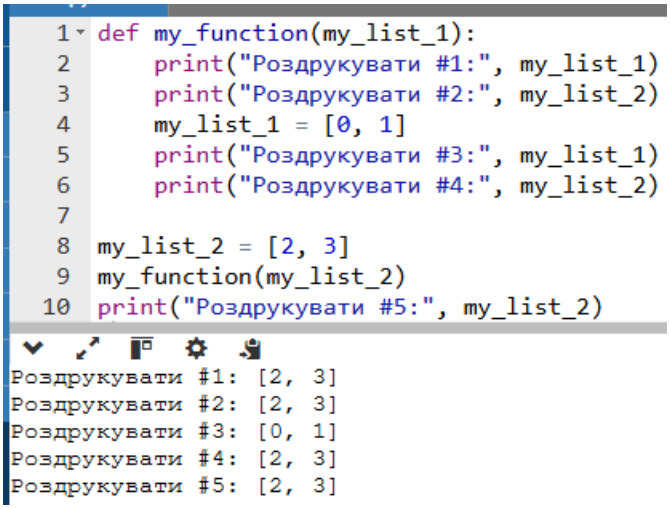


Рисунок 6.21 – Список як параметр функції

Як видно з рисунку, в нас є зовнішній список `my_list_2` та локальний `my_list_1`. `Роздрукувати #1` та `Роздрукувати #2` – виводимо вміст `my_list_1`, який містить копію `my_list_2` (передано як параметр в функцію). `Роздрукувати #3` – виводимо вміст `my_list_1` після внесених змін (рядок коду 4), `Роздрукувати #4` та `Роздрукувати #5` – виводимо зовнішній список `my_list_2`.

Розглянемо рисунок 6.22.

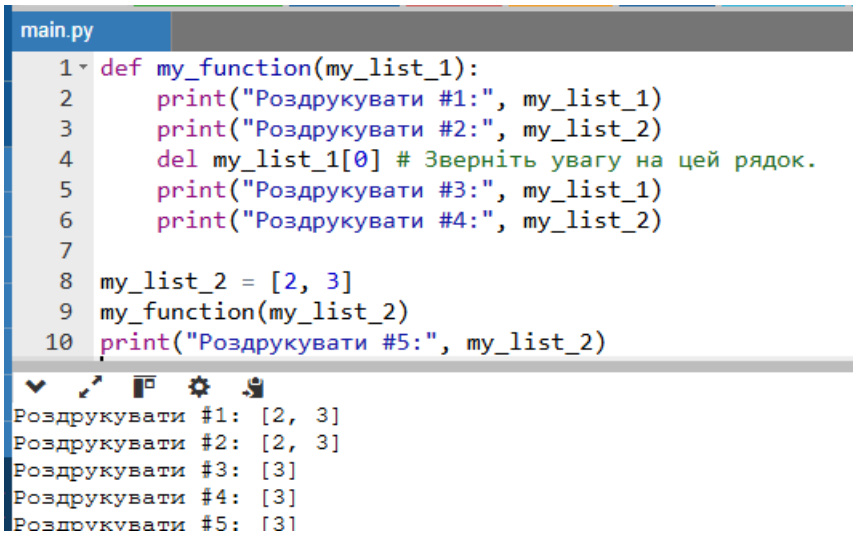


Рисунок 6.22 – Список як параметр функції

Ми не змінюємо значення параметру `my_list_1` (ми вже знаємо, що він не вплине на аргумент), а модифікуємо список, який він ідентифікує.

На основі рисунків 6.21-6.22 можемо зробити висновки:

– якщо аргумент є списком, то зміна значення відповідного параметру не впливає на список (пам’ятайте: змінні, що містять списки, зберігаються інакше, ніж скаляри);

– але якщо змінити список, визначений параметром (зверніть увагу: саме список, а не параметр!), то в списку буде відображено зміну.

6.7 Створення багатопараметричних функцій

Зі школи ми знаємо, що сума двох довільних сторін трикутника має бути більшою за третю сторону. Створимо функцію, яка матиме три параметри – по одному на кожну з сторін і поверне `True`, якщо сторони можуть утворити трикутник, і `False` якщо ні. В даному випадку, `is_a_triangle` є вдалою назвою для такої функції (рисунок 6.23).

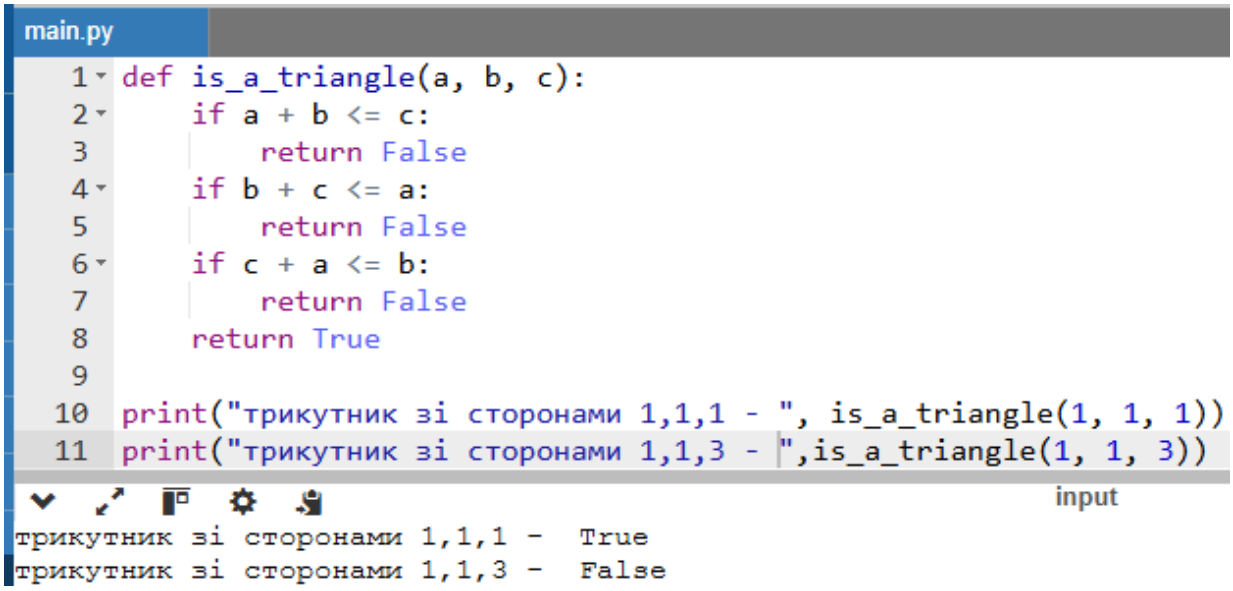


Рисунок 6.23 – Приклад функції з трьома параметрами

Тіло функції `is_a_triangle()` можна скоротити:

```
def is_a_triangle(a, b, c):
    if a + b <= c or b + c <= a or c + a <= b:
        return False
    return True
```

або:

```
def is_a_triangle(a, b, c):
    return a + b > c and b + c > a and c + a > b
```

Спробуймо переконатися, що деякий *трикутник прямокутний* (рисунок 6.24).

Потрібно буде скористатися теоремою Піфагора:

$$c^2 = a^2 + b^2$$

Звідси: гіпотенуза – це найдовша сторона.

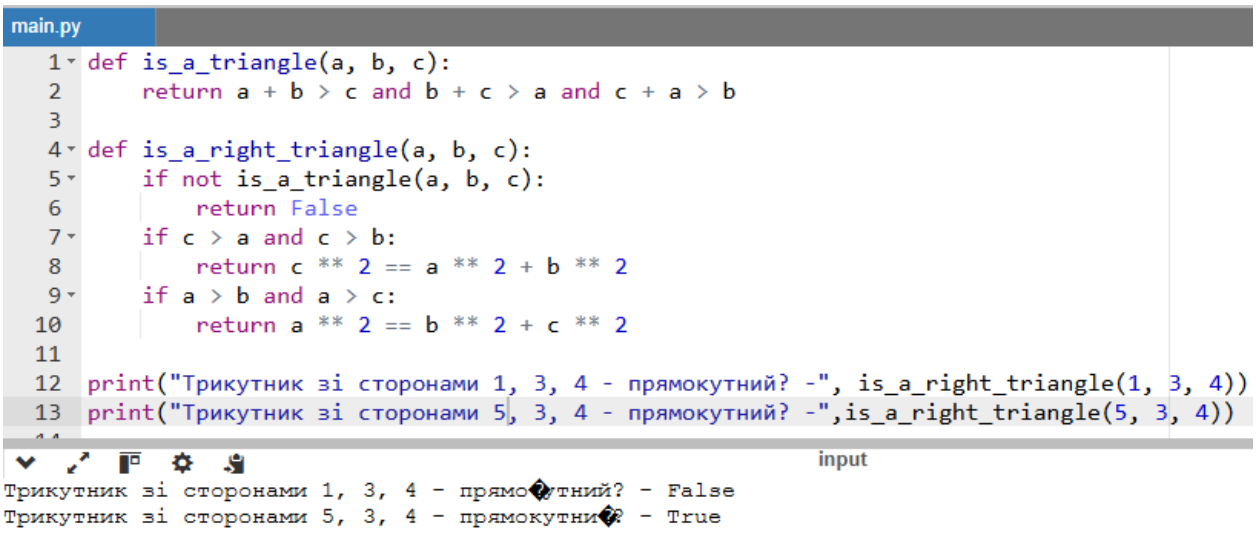


Рисунок 6.24 – Функція, що перевіряє чи трикутник прямокутний

Знайдемо площу трикутника за формулою Герона:

$$s = \frac{a + b + c}{2} \qquad A = \sqrt{s(s - a)(s - b)(s - c)}$$

Для знаходження квадратного кореня можна використовувати оператор піднесення до степеня:

$$\sqrt{x} = x^{\frac{1}{2}}$$

Приклад функцій для роботи з трикутником наведено на рисунку 6.25.

```
main.py
 1 def is_a_triangle(a, b, c):
 2     return a + b > c and b + c > a and c + a > b
 3
 4 def heron(a, b, c):
 5     p = (a + b + c) / 2
 6     return (p * (p - a) * (p - b) * (p - c)) ** 0.5
 7
 8 def area_of_triangle(a, b, c):
 9     if not is_a_triangle(a, b, c):
10         return None
11     return heron(a, b, c)
12
13 print("Площа трикутника:", area_of_triangle(1., 1., 2. ** .5))
                                                          input
Площа трикутника: 0.49999999999999983
```

Рисунок 6.25 – Знаходження площі трикутника

Пригадаймо з курсу математики що таке *факторіал*. Він позначається *знаком оклику* і дорівнює *добутку* всіх натуральних чисел від одиниці до свого аргументу.

0! = 1
1! = 1
2! = 1 * 2
3! = 1 * 2 * 3
4! = 1 * 2 * 3 * 4
…
n! = 1 * 2 * 3 * 4 * ... * n-1 * n

Створимо функцію і назвемо її `factorial_function` (рисунок 6.26).

```
main.py
 1 def factorial_function(n):
 2     if n < 0:
 3         return None
 4     if n < 2:
 5         return 1
 6     product = 1
 7     for i in range(2, n + 1):
 8         product *= i
 9     return product
10
11 for n in range(1, 6):  # тестування
12     print(n,"!=", factorial_function(n))
1 != 1
2 != 2
3 != 6
4 != 24
5 != 120
```

Рисунок 6.26 – Знаходження факторіалу

Функцію `factorial_function` можна скоротити:

```
def factorial_function(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_function(n - 1)
```

В останньому випадку використовуєтсья *рекурсія* – це коли функція викликає *сама себе*. В даному випадку:

– якщо `n =0` або `1`, то повертаємо `1` (це база рекурсії);

– інакше: `n * factorial_function(n - 1)`.

Необхідно бути дуже обережним, оскільки можна легко припуститися помилки і створити функцію, яка ніколи не завершиться. Також потрібно пам’ятати, що рекурсивні виклики займають багато місця в пам’яті, а тому іноді бувають неефективними.

Контрольні питання

1. Що таке декомпозиція задачі і чому вона важлива під час розробки програм?
2. Яке призначення функцій у Python?
3. Як оголошується функція у Python? Наведіть приклад.
4. Що таке параметризована функція?
5. Яка різниця між параметрами та аргументами функції?
6. Як передаються аргументи до функції?
7. Що означає змішування позиційних та іменованих аргументів і які правила при цьому діють?
8. Як у Python передати значення за замовчуванням для параметра функції?
9. Для чого використовується оператор return? Що означає повернення результату з функції?
10. Що повертає функція, якщо в ній відсутній оператор return або він викликається без аргументів?
11. Що таке значення None і яке його призначення у Python?
12. Що таке область видимості змінних у Python?
13. У чому відмінність між локальними та глобальними змінними?
14. Як створити багатопараметричну функцію?
15. Які переваги має використання функцій при розробці великих програмних проєктів?

ТЕМА 7. Кортежі та словники

7.1 Послідовності, типи послідовностей

У Python *послідовності* – це впорядковані колекції об’єктів, які можна перебирати, індексувати та змінювати (для змінних типів).

Тип послідовності – це тип даних у мові Python, який може зберігати більше одного значення (або менше одного, оскільки послідовність може бути порожньою), і ці значення можна переглядати послідовно (звідси і назва), елемент за елементом.

Оскільки цикл `for` є інструментом, спеціально призначеним для перебору послідовностей, то визначення можна сформулювати так: *послідовність* – це дані, які можуть бути перебрані циклом `for`.

Список є класичним прикладом послідовності на мові Python, але є й інші послідовності.

У Python існує два види даних: мінливі (змінні) та немінливі (незмінні).

Мінливі дані можуть бути вільно оновлені в будь-який час – ми називаємо таку операцію `in situ`.

`In situ` – це латинська фраза, яка перекладається дослівно як *на місці*. Наприклад, наступна інструкція модифікує дані в процесі виконання (`in situ`): `list.append(1)`

Немінливі дані не можуть бути змінені таким чином.

Уявимо, що список можна тільки призначити або перечитати, не можна ні додати до нього елемент, ні вилучити з нього будь-який елемент. Це означає, що додавання елемента в кінець списку вимагатиме створення нового списку спочатку. Тобто нам доведеться створити абсолютно новий список, який складатиметься з усіх елементів вже існуючого списку, плюс новий елемент.

Кортеж – це тип незмінної послідовності. Він може поводитися як список, але не може бути змінений в процесі виконання.

Основні типи послідовностей:

– *Списки (*`list`*)* – це змінюваний тип даних, можна додавати, видаляти, змінювати елементи; позначаються квадратними дужками `[]`. Приклад:

```
fruits = ["apple", "banana", "cherry"]
fruits[0] = "orange"
```

– *Кортежі (*`tuple`*)* – незмінюваний тип даних, використовується для зберігання фіксованих наборів, позначаються круглими дужками (). Приклад:

```
coords = (10, 20)
```

– *Рядки (*`str`*)* – незмінювана послідовність символів, можна індексувати, перебирати, але не змінювати. Приклад:

```
name = "Python"
print(name[0])  # P
```

– *Множини (*`set`*)* — технічно не є послідовністю, бо не підтримують індексацію, але часто вивчаються разом як колекції.

Загальні операції для послідовностей:

– індексація: `seq[0]`

– зрізи `(slicing): seq[1:4]`

– перебір: `for item in seq`:

– довжина: `len(seq)`

– перевірка наявності: `item in seq`

– конкатенація: `seq1 + seq2`

– повторення`: seq * 3`

Приклад використання:

```
numbers = [1, 2, 3]
numbers.append(4)
print(numbers[1:3])  # [2, 3]
```

7.2 Кортежі

Кортеж – це впорядкована, *незмінювана* послідовність елементів. Після створення його не можна змінити: не можна додати, видалити або змінити елементи.

Основні властивості:

– визначаються за допомогою круглих дужок ();

– підтримують індексацію, перебір, зрізи;

– можуть містити елементи різних типів.

Приклад:

```
# Простий кортеж
point = (10, 20)
tuple_1 = (1, 2, 4, 8) # Це кортеж
tuple_2 = 1., .5, .25, .125 # Це теж кортеж

# Кортеж з різними типами
info = ("Alice", 25, True)

# Доступ до елементів
print(point[0])  # 10

# Зрізи
print(info[1:])  # (25, True)

# Перебір
for item in info:
    print(item)

# Вкладені кортежі
matrix = ((1, 2), (3, 4))
print(matrix[1][0])  # 3
```

Якщо ми спробуємо змінити вміст кортежа, то виникне помилка:

```
person = ("John", 30)
# person[1] = 31  # Помилка! Кортежі не можна змінювати
```

В деяких випадках кортежі працюють швидше, ніж списки, крім того, вони забезпечують захист від змін (дані не зміняться випадково), їх також можна використовувати як ключі в словниках (`dict`).

Бувають кортежі, що складаються з одного елемента:

```
a = (5)     # Це просто число!
b = (5,)    # Це кортеж з одним елементом
one_element_tuple_1 = (1, )
one_element_tuple_2 = 1.,
```

Типові дії з кортежем `t`:

– `len(t)` – довжина;

– `t.count(x)` – кількість входжень `x`;

– `t.index(x)` –індекс першої появи `x`;

– оператор `+` може об’єднувати кортежі між собою;

– оператор `*` може перемножувати кортежі так само як і списки;

– оператори `in` та `not in` працюють так само, як і зі списками;

– можна розпаковувати на окремі змінні, наприклад:

```
user = ("Ann", 19)
name, age = user
print(name)  # Ann
```

Примітка: кожен елемент кортежу може бути різного типу (з рухомою крапкою, цілим чи будь-яким іншим, ще не введеним типом даних).

Якщо ми хочемо отримати елементи кортежу для того, щоб зчитати їх, то можемо скористатися тими ж правилами, до яких звикли при роботі зі списками.

Однією з корисних властивостей кортежів є їх здатність з’являтися в лівій частині оператора присвоювання. Пригадаймо приклад з попередніх лекцій, коли потрібно було здійснити перестановки місцями значення двох змінних.

Розглянемо приклад:

```
var = 123

t1 = (1, )
t2 = (2, )
t3 = (3, var)

t1, t2, t3 = t2, t3, t1

print(t1, t2, t3)
```

Він показує взаємодію трьох кортежів – по суті, значення, що зберігаються в них, «циркулюють» – `t1` стає `t2`, `t2` стає `t3`, а `t3` стає `t1`.

З цього прикладу також бачимо, що *елементами кортежу можуть бути змінні*, а не тільки літерали. Більше того, вони можуть бути виразами, якщо стоять у правій частині оператора присвоювання.

7.3 Словники

Словник – ще одна структура даних Python. Він не є послідовним типом (але може бути легко адаптований до обробки як послідовність) і є *змінним*.

Щоб пояснити, що таке словник в мові Python, важливо розуміти, що він є словником в буквальному розумінні цього слова.

Розглянемо приклад:

```
dictionary = {"cat": "chat","dog": "chien", "horse": "cheval"}
phone_numbers = {‘boss’: 5551234567, ‘Suzy’: 22657854310}
empty_dictionary = {}

print(dictionary)
print(phone_numbers)
print(empty_dictionary)
```

У `dictionary` використовуються ключі та значення, які обидва є рядками. У `phone_numbers` – ключі це рядки, а значення це цілі числа. Можлива і протилежна компоновка (ключі → цифри, значення → рядки), а також комбінація «число-число».

Перелік пар *береться у фігурні дужки*, самі пари *відокремлюються комами*, а *ключі та значення – двокрапкою*.

Перший з наших словників (`dictionary`) – це дуже простий англо-французький словник. Другий (`phone_numbers`) – дуже маленький телефонний довідник.

Останній порожній словник створений за допомогою порожньої пари фігурних дужок.

Принцип будови словника на мові Python такий самий, як і у *звичайного двомовного словника*. Наприклад, у вас є англійське слово (наприклад, `cat`) і вам потрібен його французький еквівалент. Ви шукаєте слово у словнику (можете використовувати різні техніки для цього – це не має значення) і врешті-решт знаходите його. Далі ви дивитеся на французький переклад і це ((скоріше за все) слово «`chat`».

У світі Python слово, яке ви шукаєте, називається *ключем*. Слово, яке ви знаходите у словнику, називається *значенням*.

Це означає, що словник – це набір пар *ключ-значення*. Примітка:

– кожен ключ повинен бути *унікальним* – не можна мати більше одного ключа з тим самим значенням;

– ключем може бути *будь-який незмінний тип об'єкта*: це може бути число (ціле або з рухомою крапкою), або навіть рядок, але не список;

– словник це не список – *список* містить набір *пронумерованих* значень, а *словник – пари значень*;

– функція `len()` працює і з словниками – вона повертає яка кількість елементів ключ-значення у словнику;

– словник *односторонній інструмент* – якщо у вас є англо-французький словник, ви можете шукати французькі еквіваленти англійських термінів, але не навпаки.

Розглянемо приклади. Словник повністю може бути виведений одним викликом функції `print().` Можемо отримати приблизно такий результат:

```
{‘dog’: ‘chien’, ‘horse’: ‘cheval’, ‘cat’: ‘chat’}
{‘Suzy’: 5557654321, ‘boss’: 5551234567}
{}
```

Як бачимо з останнього прикладу, порядок розташування виведених пар відрізняється від початкового.

Нагадаймо, що *словники це не списки* – вони не дотримуються порядку розташування своїх даних, оскільки цей порядок абсолютно безглуздий (на відміну від справжніх, паперових словників). Порядок, в якому словник зберігає свої дані, знаходиться поза нашим контролем, а також не відповідає нашим очікуванням. Це нормально. У Python 3.6x словники за замовчуванням стали *впорядкованими* послідовностями. Результати можуть відрізнятися залежно від того, якою версією Python ми користуємось.

Розглянемо ще один приклад (рисунок 7.1)

```
main.py
1  dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
2  phone_numbers = {'boss' : 5551234567, 'Suzy' : 22657854310}
3  empty_dictionary = {}
4  print(dictionary['cat'])
5  print(phone_numbers['Suzy'])
                                                          input
chat
22657854310
```

Рисунок 7.1 – Приклад роботи зі словником

Як бачимо з рисунку 7.1, коли ми хочемо отримати будь-яке зі значень, нам потрібно вказати дійсне значення ключа. Отримання значення зі словника нагадує пошук за індексом, особливо завдяки дужкам, що оточують значення ключа.

Примітка:

– якщо ключ є рядком, то його необхідно вказувати як рядок;

– *ключі чутливі до регістру*: `'Suzy'` – це не те ж саме, що `'suzy'`.

Не можна використовувати неіснуючий ключ. Якщо спробуємо вивести щось на зразок цього:

```
print(phone_numbers['president'])
```

то це призведе до помилки. Проте, є простий спосіб уникнути такої ситуації. Оператори `in` та `not in` можуть виправити цю ситуацію (рисунок 7.2).

```
main.py
1  dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
2  words = ['cat', 'lion', 'horse']
3
4  for word in words:
5      if word in dictionary:
6          print(word, "->", dictionary[word])
7      else:
8          print(word, "немає в словнику")
                                                          input
cat -> chat
lion немає в словнику
horse -> cheval
```

Рисунок 7.2 – Приклад використання оператора `in` для роботи зі словником

Примітка: коли ми пишемо великий або довгий вираз, краще писати словник в стовпчик. Так ми можемо зробити свій код більш читабельним та зручним для програмістів, хоча при спробі вивести на екран ми отримаємо звичайне горизонтальне виведення (рисунок 7.3). Такий вид форматування називається *висячий відступ*.

Рисунок 7.3 – Приклад покращення читабельності коду при роботі зі словником

7.4 Методи та функції роботи зі словниками

7.4.1 Метод keys()

Існують прості і дуже ефективні інструменти, які дозволяють *адаптувати будь-який словник до вимог циклу* `for` (іншими словами, побудувати проміжну ланку між словником і тимчасовою структурою послідовності.

Першим з них є метод `keys()`, який доступний для кожного словника. *Метод повертає ітерований об'єкт, що складається з усіх ключів, знайдених у словнику*. Наявність набору ключів дає змогу легко та зручно отримати доступ до всього словника (рисунок 7.4).

```
main.py
1  dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
2
3  for key in dictionary.keys():
4      print(key, "->", dictionary[key])
input
cat -> chat
dog -> chien
horse -> cheval
```

Рисунок 7.4 – Доступ до елементів словника за допомогою методу `keys()`

7.4.2 Метод items().

Метод `items()` *повертає кортежі, де кожен кортеж є парою ключ-значення* (рисунок 7.5).

```
main.py
1  dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
2
3  for english, french in dictionary.items():
4      print(english, "->", french)
input
cat -> chat
dog -> chien
horse -> cheval
```

Рисунок 7.5 – Доступ до елементів словника за допомогою методу `items()`

Зверніть увагу, що кортеж було використано як змінну циклу `for`.

7.4.3 Зміна та додавання значень

Присвоїти нове значення наявному ключу дуже просто – оскільки словники є повністю *змінюваними*, немає жодних перешкод для їхньої модифікації.

Замінимо значення "`chat`" на "`minou`", що не дуже точно, але для нашого прикладу підійде (рисунок 7.6).

```
main.py
1  dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
2
3  dictionary['cat'] = 'minou'
4  print(dictionary)
                                                              input
{'cat': 'minou', 'dog': 'chien', 'horse': 'cheval'}
```

Рисунок 7.6 – Зміна значень словника

Якщо ми хочемо відсортувати словник, то треба просто написати цикл `for`, щоб отримати таку конструкцію:

```
for key in sorted(dictionary.keys()):
```

Існує також метод з назвою `values()`, який працює аналогічно як і `keys()`, але *повертає значення* (рисунок 7.7).

```
main.py
1  dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
2
3  for french in dictionary.values():
4      print(french)
                                                              input
chat
chien
cheval
```

Рисунок 7.7 – Приклад використання методу `values()`

Оскільки словник не в змозі автоматично знайти ключ для заданого значення, роль цього методу досить обмежена.

7.4.4 Додавання нового ключа

Додати нову пару ключ-значення до словника так само просто, як і змінити значення – потрібно лише присвоїти значення новому, *раніше неіснуючому ключу*.

Примітка: це зовсім інша поведінка порівняно зі списками, які не дозволяють присвоювати значення неіснуючим індексам.

Додамо до словника нову пару слів (рисунок 7.8).

Рисунок 7.8 – Приклад додавання даних до словника

Ми також можемо вставити елемент у словник за допомогою методу `update()` (рисунок 7.9).

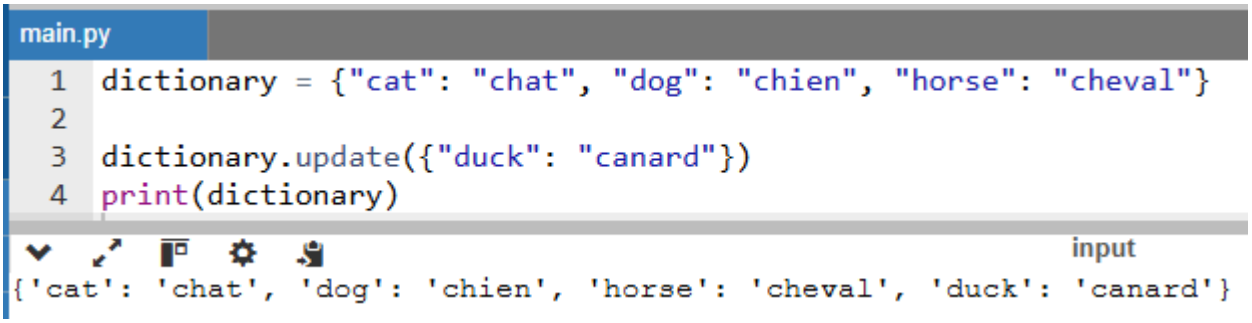


Рисунок 7.9 – Приклад використання методу `update()`

7.4.5 Видалення ключа

Видалення ключа завжди призводить до видалення пов’язаного з ним значення. *Значення не можуть існувати без своїх ключів*.

Це робиться за допомогою інструкції `del` (рисунок 7.10).

Рисунок 7.10 – Приклад видалення даних зі словника

Примітка: видалення неіснуючого ключа призведе до помилки.

Для видалення останнього елементу словника можна використати метод `popitem()` (рисунок 7.11). У старих версіях Python, до версії 3.6.7, метод `popitem()` видаляв випадковий елемент зі словника.

Рисунок 7.11 – Приклад видалення останнього значення зі словника

Основні методи для роботи зі словниками наведено в таблиці 7.1.

Таблиця 7.1 – Методи роботи зі словниками

Метод	Опис
`dict.keys()`	Повертає список ключів
`dict.values()`	Повертає список значень
`dict.items()`	Повертає список пар (ключ, значення)
`dict.get(key, default)`	Повертає значення за ключем або default, якщо ключа немає
`dict.pop(key)`	Видаляє пару за ключем і повертає значення
`dict.update(other_dict)`	Оновлює словник з іншого словника або пари ключів
`dict.clear()`	Очищає словник
`key in dict`	Перевіряє наявність ключа у словнику

7.5 Взаємодія кортежів та словників

Уявімо собі наступну проблему:

– потрібна програма для визначення середньої оцінки студентів;

– програма повинна запитувати ім’я студента, а потім його особисту оцінку;

– імена можна вводити в будь-якому порядку;

– введення порожнього імені завершує введення;

– після цього має бути виданий список усіх імен разом з визначеним середнім балом.

Розглянемо приклад на рисунку 7.12.

```
1  school_class = {}
2
3  while True:
4      name = input("Введіть ім'я студента: ")
5      if name == '':
6          break
7
8      score = int(input("Введіть оцінку студента (0-100): "))
9      if score not in range(0, 101):
10             break
11
12     if name in school_class:
13         school_class[name] += (score,)
14     else:
15         school_class[name] = (score,)
16
17 for name in sorted(school_class.keys()):
18     adding = 0
19     counter = 0
20     for score in school_class[name]:
21         adding += score
22         counter += 1
23     print(name, ":", adding / counter)
```

Рисунок 7.12 – Приклад взаємодії кортежів та словників

Проаналізуємо його рядок за рядком:

– *рядок 1*: створюємо порожній словник для вхідних даних; в якості ключа використовується ім'я студента, а всі пов'язані з ним оцінки зберігаються в кортежі (кортеж може бути значенням словника – це не є проблемою);

– *рядок 3*: створюємо «нескінченний» цикл (не хвилюйтеся, він буде перерваний в потрібний момент);

– *рядок 4*: прочитайте тут ім'я студента;

– *рядок 5-6*: якщо ім'я є порожнім рядком (), вийдемо з циклу;

– *рядок 8*: запитується одна з оцінок студента (ціле число з діапазону 0-100)

– *рядок 9-10*: якщо введена оцінка не знаходиться в діапазоні від 0 до 100, вийти з циклу;

– *рядок 12-13*: якщо прізвище студента вже є в словнику, подовжити пов'язаний кортеж з новою оцінкою (зверніть увагу на оператор +=);

– *рядок 14-15*: якщо це новий студент (невідомий словнику), створити новий запис – його значенням буде одноелементний кортеж з введеною оцінкою;

– *рядок 17*: перебрати відсортовані імена студентів;

– *рядок 18-19*: ініціалізація даних, необхідних для обчислення середньої оцінки (сума та лічильник);

– *рядок 20-22*: виконуємо ітерації по кортежу, беручи всі наступні оцінки і оновлюючи суму разом з лічильником;

– *рядок 23*: виводимо ім’я студента та його середню оцінку.

Результат роботи програми наведено на рисунку 7.13.

```
Введіть ім'я студента: Ірина
Введіть оцінку студента (0-100): 100
Введіть ім'я студ♦нта: Микола
Введіть оцінку студента (0-100): 85
Введіть ім'я студ♦нта: Ірина
Введіть оцінку студента (0-100): 70
Введіть ім'я студ♦нта: Іван
Введіть оцінку студента (0-100): 65
Введіть ім'я студ♦нта: Микола
Введіть оцінку студента (0-100): 40
Введіть ім'я студ♦нта:
Іван : 65.0
Ірина : 85.0
Микола : 62.5
```

Рисунок 7.13 – Протокол виконання програми

Контрольні питання

1. Що таке послідовність (sequence) у Python і які типи послідовностей ви знаєте?
2. Чим кортеж відрізняється від списку?
3. Як створити кортеж у Python? Наведіть приклади різних способів створення.
4. Чи можна змінювати елементи кортежу після його створення? Поясніть чому.
5. Як отримати доступ до елементів кортежу за індексом?
6. Як виконати розпакування кортежів? Наведіть приклад.
7. У яких випадках доцільно використовувати кортеж замість списку?
8. Що таке словник у Python і для чого він використовується?
9. Як створити словник? Наведіть приклади різних способів створення словників.
10. Як звернутися до значення в словнику за ключем?
11. Як додати нову пару ключ-значення до словника або змінити існуючу?
12. Які основні методи словників ви знаєте?
13. Як можна перебрати всі ключі й значення словника в циклі for?

ТЕМА 8. Робота з модулями та пакетами

8.1 Поняття про модулі

Модуль – це файл із Python-кодом (з розширенням `.py`), який містить функції, класи, змінні або навіть цілі програми, і який можна імпортувати та використовувати в інших програмах.

Шукати помилки завжди легше там, де код менший (подібно до того, як знайти механічну поломку легше, коли механізм простіший і менший).

Разом із самим Python поставляється (досить велика) кількість модулів.

Усі ці модулі разом із вбудованими функціями утворюють *стандартну бібліотеку Python* – особливий вид бібліотеки, де модулі виконують роль книг (можна навіть сказати, що папки виконують роль полиць). Якщо ми хочемо переглянути повний список усіх «томів», зібраних у цій бібліотеці, то можемо знайти його тут: https://docs.python.org/3/library/index.html.

Кожен модуль складається з сутностей (як книга складається з розділів). Ці сутності можуть бути функціями, змінними, константами, класами та об'єктами. Якщо ми знаємо, як отримати доступ до певного модуля, то можемо використовувати будь-які сутності, які він зберігає.

Модулі потрібні щоб:

– організувати код: розбивати великі програми на частини;

– повторно використовувати код без копіювання;

– використовувати готові бібліотеки (наприклад, `math`, `random`, `datetime`).

Наприклад, ми можемо створити файл `pl_math.py` і оголосити в ньому функції для обчислення факторіалу, степеня, кореня, чисел Фібоначчі та інші. Далі ми можемо створити нову програму і підключити до неї модуль `pl_math` (він має знаходитися в одній папці з програмою, ім'я модуля збігається з іменем файлу, лише без розширення `.py`) і використовувати всі ці функції в новій програмі.

Як бачите, ще однією перевагою модулів є те, що оголошення, зроблені в модулі, можуть використовуватися не в одній, а в будь-якій програмі, яку ми писатимемо далі.

8.2 Імпортування та використання модулів. Простір імен

Щоб зробити модуль придатним для використання, ми повинні його *імпортувати* (це як взяти книгу з полиці). Імпортування модуля виконується інструкцією `import` (це ключове слово).

Припустімо, що ми хочемо використовувати дві сутності, надані в модулі `math`:

– символ (константу), що містить якомога точніше значення π (позначається як `pi`);

– функцію `sin()`.

Обидві ці сутності доступні через модуль `math`, але спосіб їх використання залежить від того, як було виконано імпорт.

Найпростіший спосіб імпортувати окремий модуль – це використати наступну інструкцію імпорту:

```
import math
```

Такий рядок можна розташувати будь-де у нашому коді, але її потрібно розмістити перед першим використанням будь-якої сутності модуля.

Якщо ми хочемо імпортувати більше одного модуля, то можемо зробити це так (бажано):

```
import math
import sys
```

Або так:

```
import math, sys
```

Список модулів може бути як завгодно довгим.

Простір імен – це простір (логічний, а не фізичний), у якому існують деякі імена, які не конфліктують між собою (тобто немає двох різних об'єктів з однаковою назвою). Ми можемо сказати, що кожна соціальна група є простором імен – група прагне називати кожного свого члена унікальним способом (наприклад, батьки не дадуть своїм дітям однакові імена).

Цієї унікальності можна досягти багатьма способами, наприклад, використовуючи псевдоніми разом з іменами (це буде працювати всередині невеликої групи, як у класі в школі) або шляхом призначення спеціальних ідентифікаторів усім членам групи (наприклад, ідентифікаційний код в податковій).

У певному просторі імен кожне ім'я має залишатися унікальним. Це може означати, що деякі імена можуть зникнути, коли будь-яка інша сутність із уже відомою назвою входить у простір імен.

Якщо модуль із вказаною назвою існує та є доступним (модуль насправді є вихідним файлом Python), Python імпортує його вміст, тобто всі імена, визначені в модулі, стають відомими, але вони не входять у простір імен нашого коду.

Це означає, що ми можемо мати власні назви сутностей `sin()` або `pi`, імпорт жодним чином не вплине на них.

Щоб отримати доступ до `pi`, що походить від модуля `math` нам потрібно вказати назву відповідного модуля, як на рисунку 8.1.

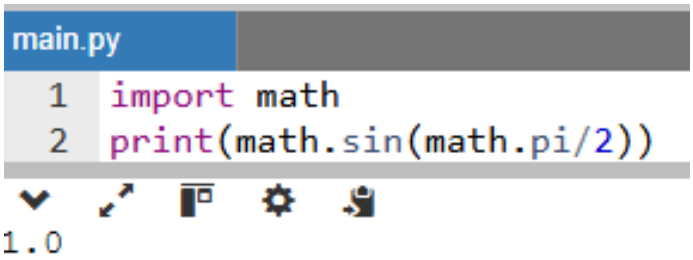


Рисунок 8.1 – Приклад використання сутностей з модуля `math`

Отже, щоб використати якусь сутність модуля, потрібно написати:

– назву модуля (наприклад, `math`);

– крапку (тобто `.`);

– назву сутності (наприклад, `pi`).

Така форма чітко вказує простір імен, у якому існує назва сутності.

Розглянемо як можуть співіснувати два простори імен (наш і модульний) – рисунок 8.2.

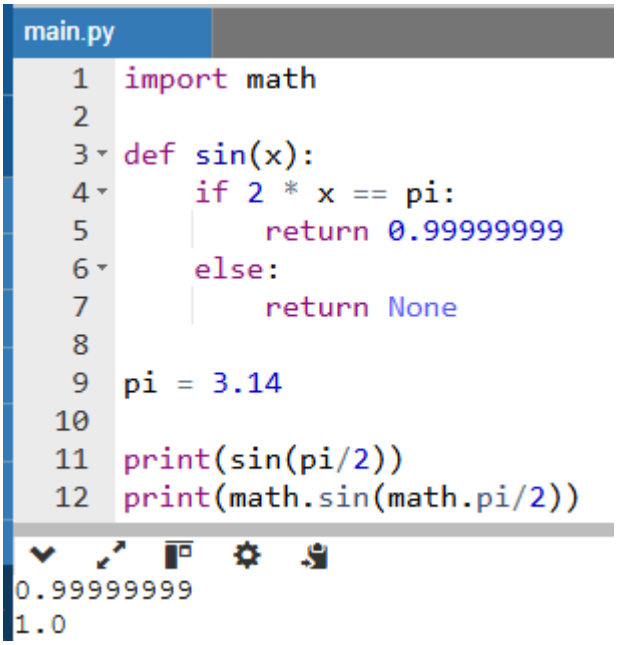


Рисунок 8.2 – Приклад використання двох просторів імен

Як бачимо, сутності не впливають одна на одну.

Другий спосіб імпорту:

```
from math import pi
```

Ця інструкція складається з наступних елементів:

– ключове слово `from`;

– ім’я модуля , який (вибірково) імпортується (в прикладі `math`);

– ключове слово `import`;

– ім’я або список імен сутності/сутностей, які імпортуються в простір імен (в прикладі `pi`).

Це має такий ефект:

– лише перераховані сутності імпортуються з указаного модуля;

– назви імпортованих об’єктів доступні без застережень (рисунок 8.3).

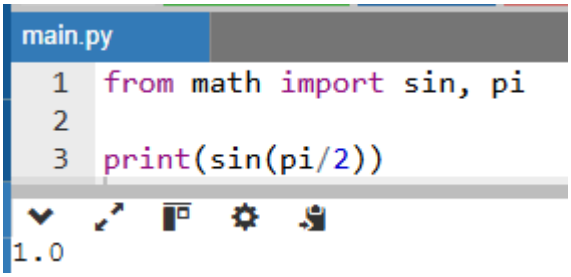


Рисунок 8.3 – Приклад імпортування частини модуля

Третій спосіб імпорту:

```
from module import *
```

Тут * позначено всі сутності модуля. При такому варіанті імпорту нам не потрібно перераховувати всіх сутностей, але якщо ми не знаємо всіх назв, наданих модулем, то не зможемо уникнути конфліктів імен.

Якщо ми підключимо до основної програми і `math`, і `pl_math`, у нас буде дві функції з однаковими іменами. Але це не призведе до помилки, так як вони визначені в різних просторах імен: `pl_math.exp` та `math.exp` і при зверненні до них інтерпретатор чітко знатиме, яку саме ми маємо на увазі.

Зверніть увагу: для підключення інтерпретатор шукає модулі у поточній папці. Якщо ми підключаємо модуль всередині файлу (програми або іншого модулю) –це папка, в якій знаходиться цей файл. Якщо ми підключаємо модуль в інтерактивному режимі інтерпретатора – це папка, в якій його було запущено. Якщо в поточній папці файл модуля не знайдено, інтерпретатор перевіряє деякий шлях, вказаний в налаштуваннях Python (відрізняється для різних операційних систем). Саме там знаходяться більшість стандартних модулів, туди ж ми можемо додавати власні модулі, туди ж можна додавати модулі, знайдені в інтернеті, для використання в своїх програмах.

8.3 Псевдоніми

Якщо ми використовуємо перший варіант імпорту і нам не подобається ім’я певного модуля (наприклад, воно збігається з однією з наших уже визначених

сутностей), ми можемо дати йому будь-яке ім’я, яке нам подобається – це називається *псевдонімом*.

Псевдоніми призводять до того, що модуль ідентифікується під назвою, відмінною від оригінальної. Це також може використовуватися щоб скоротити оригінальні імена.

Створення псевдоніма виконується разом з імпортом модуля та вимагає такої форми інструкції імпорту:

```
import module as alias
```

`module` визначає ім’я вихідного модуля, а `alias` – це ім’я, яке ми хочемо використовувати замість оригіналу, `as` – ключове слово.

Якщо потрібно змінити слово `math`, ми можемо ввести власне ім’я, як на рисунку 8.4.

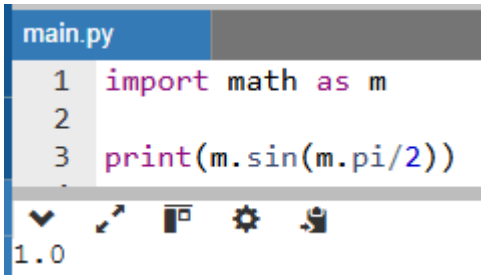


Рисунок 8.4 – Приклад використання псевдоніма

Примітка: після успішного виконання псевдонімного імпорту оригінальне ім’я модуля стає недоступним і його не можна використовувати.

Якщо ми використовуємо другий варіант імпорту і нам потрібно змінити назву сутності, ми створюємо псевдонім для сутності. Це призведе до того, що ім’я буде замінено вибраним нами псевдонімом.

Ось як це можна зробити:

```
from module import name as alias
```

Як і раніше, вихідне (без псевдонімів) ім’я стає недоступним.

Можна одразу через коми викначити кілька псевдонімів (рисунок 8.5), наприклад:

```
from module import n as a, m as b, o as c
```

```
main.py
1  from math import pi as PI, sin as sine
2
3  print(sine(PI/2))
1.0
```

Рисунок 8.5 – Приклад використання кількох псевдонімів

8.4 Робота зі стандартними модулями

8.4.1 Функція dir()

Функція `dir()` повертає впорядкований за алфавітом список, що містить усі імена об'єктів, доступні в модулі, визначені іменем, переданим функції як аргумент:

```
dir(module)
```

Примітка: якщо назву модуля було перейменовано псевдонімом, ми повинні використовувати псевдонім, а не оригінальну назву.

Щоб дізнатися які функції містить модуль, можна виконати код як на рисунку 8.6 або на рисунку 8.7.

```
main.py
1  import math
2
3  for name in dir(math):
4    print(name, end="\t")
                                                    input
__doc__  __loader__      __name__        __package__     __spec__
tanh     cbrt    ceil    comb    copysign        cos     cosh    degrees
xpm1     fabs    factorial       floor   fmod    frexp   fsum    gamma
sinf     isnan   isqrt   lcm     ldexp   lgamma  log     log10   log1p
i        pow     prod    radians remainder       sin     sinh    sqrt
```

Рисунок 8.6 – Перелік функцій модуля `math`

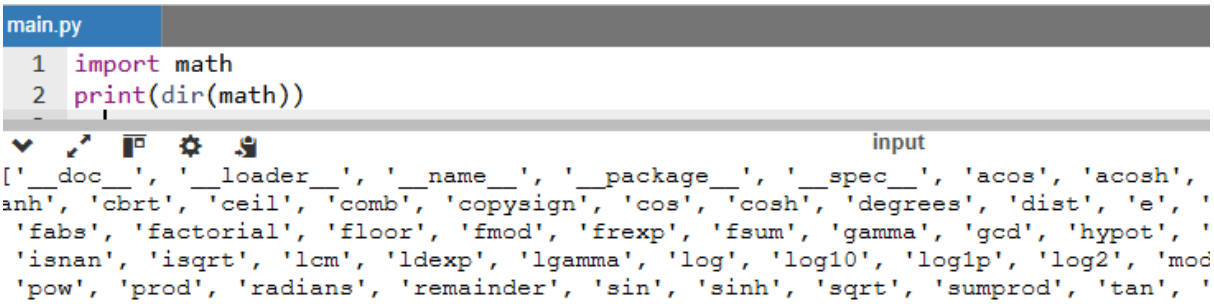


Рисунок 8.7 – Вміст модуля `math`

9.4.2 Модуль math

Модуль `math` надає набір математичних функцій і констант для роботи з числами (поєднує понад 50 символів, функцій і констант). Деякі функції модуля `math` наведено в таблицях 8.1-8.3.

Таблиця 8.1 – Основні функції модуля `math`

Функція	Призначення	Приклад
`math.sqrt(x)`	Квадратний корінь	`math.sqrt(16) → 4.0`
`math.pow(x, y)`	Піднесення до степеня	`math.pow(2, 3) → 8.0`
`math.exp(x)`	Експонента (e^x)	`math.exp(1) → 2.71...`
`math.log(x[, base])`	Логарифм (натуральний або з основою)	`math.log(8, 2) → 3.0`
`math.factorial(x)`	Факторіал (x!)	`math.factorial(5) → 120`
`math.floor(x)`	Округлення вниз	`math.floor(3.9) → 3`
`math.ceil(x)`	Округлення вгору	`math.ceil(3.1) → 4`
`math.fabs(x)`	Абсолютне значення	`math.fabs(-5) → 5.0`
`math.isqrt(x)`	Цілий квадратний корінь	`math.isqrt(10) → 3`

Таблиця 8.2 – Тригонометричні функції

Функція	Приклад
`math.sin(x)`	`math.sin(math.pi / 2) → 1.0`
`math.cos(x)`	`math.cos(0) → 1.0`
`math.tan(x)`	`math.tan(math.pi / 4) → ~1.0`
`math.degrees(x)`	Перетворення з радіан в градуси
`math.radians(x)`	Перетворення з градусів у радіани

Таблиця 8.3 – Константи

Назва	Значення
`math.pi`	`3.1415926535…`
`math.e`	`2.7182818284…`
`math.tau`	`6.2831853071… (2π)`
`math.inf`	Нескінченність
`math.nan`	Не число ("Not a Number")

Приклад практичного використання:

```
import math

r = 5
area = math.pi * math.pow(r, 2)
print(f"Площа круга з радіусом {r}: {area}")
```

Приклад використання функцій заокруглення наведено на рисунку 8.8.

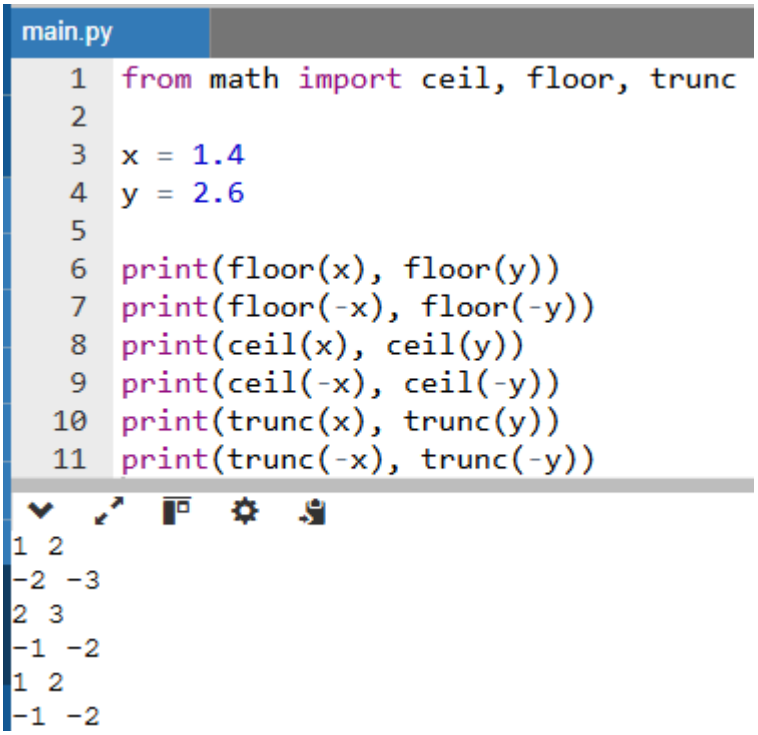


Рисунок 8.8 – Приклад використання функцій заокруглення з модуля `math`

8.4.3 Модуль random

Модуль `random` використовується для генерації псевдовипадкових чисел та випадкового вибору елементів зі списків, рядків тощо, він групує понад 60 сутностей.

Алгоритми не є випадковими – вони детерміновані та передбачувані. Лише ті фізичні процеси, які повністю виходять з-під нашого контролю (наприклад, інтенсивність космічного випромінювання), можуть бути використані як джерело фактичних випадкових даних. Дані, створені детермінованими комп’ютерами, жодним чином не можуть бути випадковими.

Генератор випадкових чисел приймає значення, яке називається початковим значенням, розглядає його як вхідне значення, обчислює на його основі

«випадкове» число (метод залежить від вибраного алгоритму) і створює нове початкове значення.

Тривалість циклу, в якому всі вихідні значення унікальні, може бути дуже довгою, але вона не нескінченна – рано чи пізно вихідні значення почнуть повторюватися, і генеруючі значення також повторюватимуться.

Отож, модуль `random` дозволяє працювати з псевдовипадковими числами (таблиця 8.4).

Таблиця 8.4 – Функції модуля `random`

Функція	Опис
`random.random()`	Випадкове число від 0.0 до 1.0 (`float`)
`random.randint(a, b)`	Ціле число від `a` до `b` (включно)
`random.randrange(a, b[, step])`	Випадкове ціле з діапазону (як `range`)
`random.uniform(a, b)`	Випадкове дійсне число від `a` до `b`
`random.choice(seq)`	Випадковий елемент зі списку/рядка
`random.choices(seq, k=n)`	Список з `n` випадкових елементів (з повторами)
`random.sample(seq, k=n)`	Список з `n` випадкових елементів (без повторів)
`random.shuffle(seq)`	Перемішує список на місці

Приклади використання цього модуля наведено на рисунках 8.9-8.12.

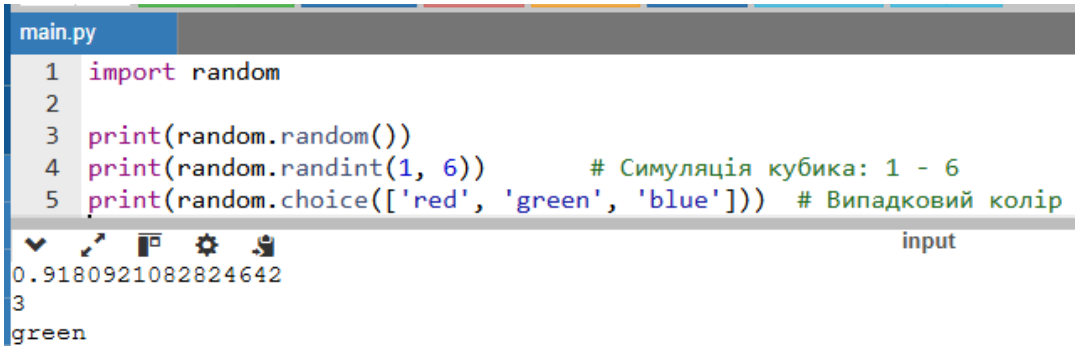


Рисунок 8.9 – Приклад використання функцій з модуля `random`

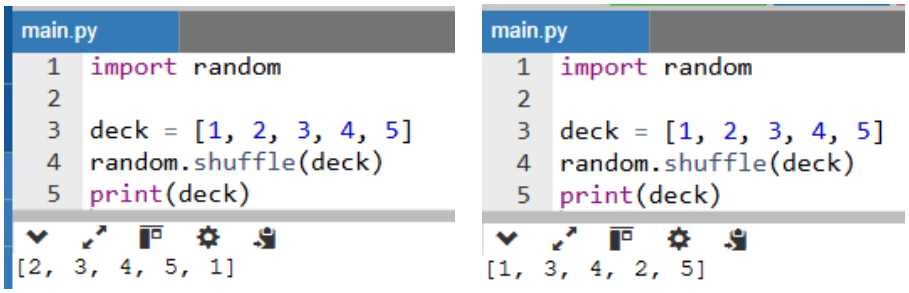


Рисунок 8.10 – Приклад випадкового перемішування списку

main.py

```
1  import random
2
3  random.seed(42)  # Встановлює "зерно" для повторюваних результатів
4  print(random.random())  # Завжди однаковий результат при однаковому seed
```

input

```
0.6394267984578837
```

Рисунок 8.11 – Випадковість, яка повторюється

main.py

```
1  from random import choice, sample
2
3  my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4
5  print(choice(my_list))
6  print(sample(my_list, 5))
7  print(sample(my_list, 10))
```

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

Рисунок 8.12 – Приклад використання функцій `choice` та `sample`

Цей модуль часто застосовується при створенні ігор (випадкові події, кидки кубика), для генерації паролів, при моделюванні випадкових подій, перемішуванні елементів тощо.

8.4.4 Модуль platform

Модуль `platform` надає функції для отримання *технічної інформації про систему*, на якій виконується Python-скрипт, він містить близько 70 функцій.

Іноді може знадобитися дізнатися інформацію, не пов’язану з Python. Наприклад, нам може знадобитися розташування нашої програми в загальному середовищі комп’ютера (рисунок 8.13).

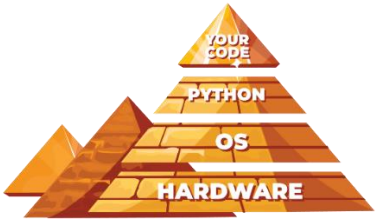


Рисунок 8.13 – Піраміда рівнів ПЗ в комп’ютері

Можна уявити середовище нашої програми як піраміду, що складається з кількох рівнів або платформ:

– наш (запущений) код розташований у верхній частині;

– Python (точніше – його середовище виконання) лежить безпосередньо під ним;

– наступний рівень піраміди заповнює ОС (операційна система) – середовище Python надає деякі свої функції за допомогою служб операційної системи; хоча Python дуже потужний, він не всемогутній – він змушений використовувати багато помічників, якщо він збирається обробляти файли або спілкуватися з фізичними пристроями;

– найнижчий рівень – апаратне забезпечення – процесор (або процесори), мережеві інтерфейси, пристрої людського інтерфейсу (миші, клавіатури тощо) та всі інші механізми, необхідні для роботи комп'ютера; ОС знає, як керувати цим всім.

Це означає, що деякі наші дії (точніше нашої програми) повинні пройти довгий шлях, щоб бути успішно виконаними. Нехай:

– наш код хоче створити файл, тому він викликає одну з функцій Python;

– Python приймає порядок, змінює його відповідно до вимог локальної ОС (це як поставити штамп «схвалено» на вашому запиті) і надсилає його вниз (це може нагадувати вам ланцюжок команд)

– ОС перевіряє, чи запит обґрунтований і дійсний (наприклад, чи відповідає ім'я файлу деяким правилам синтаксису) і намагається створити файл; така операція, яка здається дуже простою, не є атомарною – вона складається з багатьох незначних кроків, які виконує апаратне забезпечення, яке відповідає за активацію пристроїв зберігання (жорсткий диск тощо) для задоволення потреб ОС.

Зазвичай ми не усвідомлюємо цього, ми просто хочемо, щоб файл був створений, і все. Але іноді програміст хочете знати більше, наприклад, назву ОС, на якій розміщено Python, і деякі характеристики, що описують апаратне забезпечення, на якому розміщено ОС. Для цього можна використати модуль `platform` (таблиця 8.5).

Таблиця 8.5 – Корисні функції модуля `platform`

Функція	Опис та приклад виводу
`platform.system()`	Назва ОС: ‘Windows’, ‘Linux’, ‘Darwin’
`platform.release()`	Версія релізу ОС
`platform.version()`	Детальна версія ОС
`platform.machine()`	Тип машини (напр. ‘x86_64’)
`platform.processor()`	Назва процесора (напр. ‘Intel64 Family 6’)
`platform.platform()`	Повний рядок опису ОС
`platform.node()`	Ім’я комп’ютера (мережеве ім’я)
`platform.python_version()`	Версія Python (напр. ‘3.10.12’)

Приклад використання функцій цього модуля наведено на рисунку 8.14.

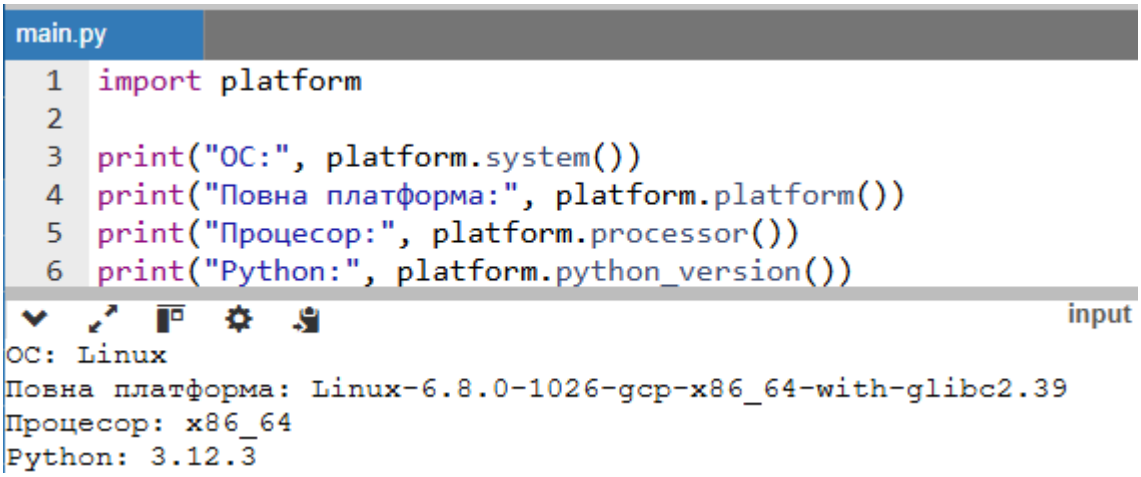


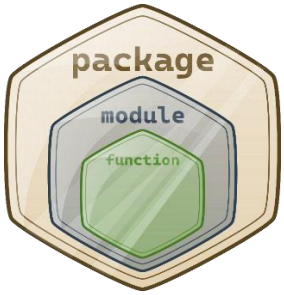
Рисунок 8.14 – Приклад використання функцій модуля `platform`

Такі функції використовуються для знаходження інформації про середовище для логів та діагностики, при створенні кросплатформеного ПЗ, при автоматичному визначенні ОС для запуску відповідного коду, тощо.

Тут ми розглянули лише основи модулів Python. Спільнота Python у всьому світі створює та підтримує сотні додаткових модулів, які використовуються в дуже нішевих програмах, таких як генетика, психологія чи навіть астрологія. Ці модулі не поширюються (і не будуть) розповсюджуватися разом із Python або через офіційні канали. Про всі стандартні модулі Python можна прочитати про тут: https://docs.python.org/3/py-modindex.html .

8.5 Пакети. Впорядкування програм

Коли модулів стає забагато, виникає необхідність групувати їх далі. Для цього файли модулів розкладаються по папках. Ми знаємо, що інтерпретатор шукає модулі в поточній папці та у спеціально призначеному для цього місці, отже необхідно якимось чином показати йому, що папка поряд з нашою програмою – не просто папка з файлами, а вона містить модулі для підключення. Для цього в папці повинен знаходитися файл `__init__.py` – він може бути порожнім, але сама його наявність сигналізує інтерпретатору, що папка із ним є пакетом модулів і може використовуватися в програмі (рисунок 8.15).




```
my_package/
│
├── __init__.py
├── math_utils.py
└── string_utils.py
```

Рисунок 8.15 – Структура пакета

Як і модулі, пакети створюють нові простори імен:

```
import my_pl_package.pl_math # модуль pl_math
#шукатиметься в пакеті my_pl_package (в одноіменній папці)
print my_pl_package.pl_math.exp(1)
```

Все, що говорилося про види імпорту поширюється також на пакети. Лише в іменах додається додатковий елемент через крапку – назва пакету. Пакети можуть вкладатися в інші пакети, аналогічно додаючи нові простори імен. Подальше структурування програми обмежується лише нашою фантазією.

Як і модулі, пакети можуть містити код, який буде виконано під час ініціалізації пакету, – він записується в самому файлі `__init__.py`.

Модуль – це свого роду контейнер, наповнений функціями – ми можемо упакувати скільки завгодно функцій в один модуль і розповсюдити його (за потреби).

Як правило, не варто змішувати функції з різними областями застосування в одному модулі (так само, як у бібліотеці – ніхто не очікує, що наукові роботи будуть розміщені серед коміксів), тому потрібно ретельно згрупувати свої функції та називати модуль, який їх містить, чітко та інтуїтивно зрозуміло (наприклад, не вказуйте назву arcade_games до модуля, що містить функції, призначені для розділення та форматування жорстких дисків).

Пакети використовують:

– у великих проектах для структурування коду;

– для створення бібліотек і фреймворків;

– для розмежування логіки (напр. робота з файлами, мережею, БД).

Приклад структури проєкту, що використовує власний пакет, наведено на рисунку 8.16.

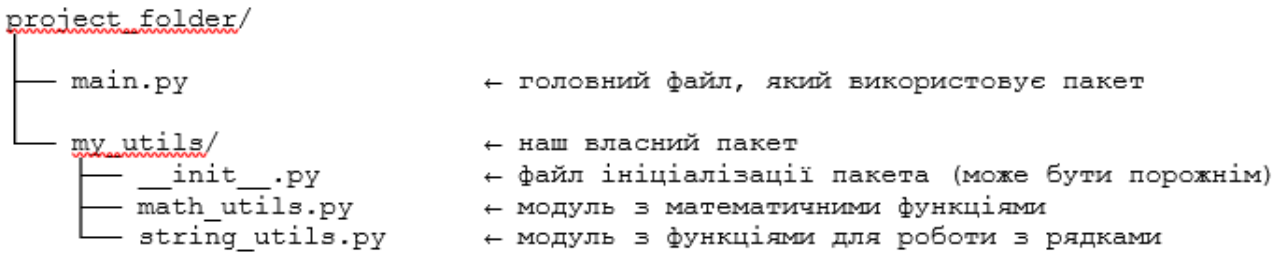


Рисунок 8.16 – Структура проєкту з власним пакетом

Файл `main.py`:

```
from my_utils import math_utils, string_utils

print(math_utils.add(2, 3))             # 5
print(math_utils.multiply(4, 5))        # 20

print(string_utils.to_upper("hello"))  # HELLO
print(string_utils.reverse("world"))   # dlrow
```

Файл `my_utils/init.py`:

```
# Пакет корисних утиліт
```

Файл `my_utils/math_utils.py`:

```
def add(a, b):
```

```
    return a + b

def multiply(a, b):
    return a * b
```

Файл `my_utils/string_utils.py`:

```
def to_upper(text):
    return text.upper()

def reverse(text):
    return text[::-1]
```

Якщо зазирнемо в папку `my_utils`, то побачимо, що там з'явилася ще одна папка – `__pycache__` (рисунок 8.17).

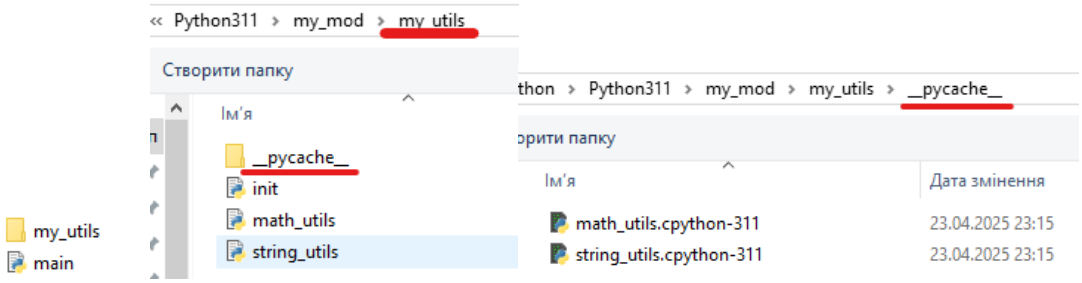


Рисунок 8.17 – Папки та файли проєкту

Якщо зазирнути всередину файлу (наприклад, `math_utils.cpython-311`) – виявимо вміст абсолютно нерозбірливий для людини. Так і має бути, оскільки файл призначений лише для використання Python.

Коли Python імпортує модуль вперше, він перетворює його вміст у *напівскомпільовану* форму. Цей файл не містить машинного коду – це внутрішній напівскомпільований код Python , готовий до виконання інтерпретатором Python. Таким чином, файл не вимагає багато перевірок, необхідних для чистого вихідного файлу, виконання починається швидше, і він також працює швидше.

Завдяки цьому кожен наступний імпорт відбуватиметься швидше, ніж інтерпретація вихідного тексту з нуля.

Якщо ми хочемо повідомити користувачеві нашого модуля, що певну сутність слід розглядати як приватну (тобто не використовувати явно поза

модулем), ми можемо позначити її ім’я префіксом _ або __. Проте це лише рекомендація (умовна позначка для програмістів), інтерпретатор за цим слідкувати не буде.

8.6 Екосистема пакетів Python

Python став міждисциплінарним інструментом, який використовується в багатьох програмах.

Перш за все, Python став лідером досліджень *штучного інтелекту*. Інтелектуальний аналіз даних, одна з найперспективніших сучасних наукових дисциплін, також використовує Python. Математики, психологи, генетики, метеорологи, лінгвісти – усі ці люди вже використовують Python. Від цієї тенденції нікуди не дітися.

Немає сенсу змушувати всіх користувачів Python писати свій код з нуля, тримаючи їх у повній ізоляції від зовнішнього світу та досягнень інших програмістів. Це було б і неприродно, і непродуктивно.

Найбільш ефективним є надання всім членам спільноти Python вільного обміну кодами та досвідом. У цій моделі ніхто не змушений починати роботу з нуля, оскільки існує висока ймовірність того, що хтось інший працював над тією ж (або дуже схожою) проблемою.

Як ви знаєте, Python було створено як програмне забезпечення з відкритим кодом, і це також спонукає програмістів підтримувати всю екосистему Python як відкрите, дружнє та вільне середовище. Щоб модель працювала та розвивалася, слід надати деякі додаткові інструменти, які допомагають творцям публікувати, підтримувати та піклуватися про свій код.

Екосистема Python-пакетів – це величезна множина готових бібліотек, модулів та фреймворків, які розширюють можливості Python:

– робота з даними;
– візуалізація;
– розробка веб-сайтів;
– машинне навчання;

– і багато іншого.

Найбільше сховище пакетів – PyPI (Python Package Index) – це онлайн-сховище, де розміщуються десятки тисяч готових пакетів. Сайт: https://pypi.org.

Слід зазначити, що PyPI – не єдине існуюче сховище Python. Навпаки, їх багато, створених для проектів і керованих багатьма більшими та меншими спільнотами Python. Можливо, колись ви чи ваші колеги захочете створити власні репозиторії.

На даний час PyPI є найважливішим репозитарієм Python у світі. PyPI абсолютно безкоштовний.

Щоб встановити сторонній пакет використовується утиліта `pip` (Python Installer Package):

```
pip install назва_пакета
```

Приклад використання цієї утиліти наведено на рисунку 8.18.

main.py

```
1  pip install requests        # Для HTTP-запитів
2  pip install matplotlib      # Для графіків
3  pip install numpy           # Для обчислень з масивами
```

Рисунок 8.18 – Приклад встановлення сторонніх пакетів

Перевірити які є встановлені пакети:

```
pip list
```

Видалити пакет:

```
pip uninstall назва_пакета
```

Деякі інсталяції Python постачаються з `pip` , а деякі ні.

Інсталятор MS Windows Python уже містить `pip` , тому для його встановлення не потрібно виконувати жодних інших кроків. На жаль, якщо змінну `PATH` налаштовано неправильно, `pip` може бути недоступним.

Щоб переконатися що `pip` встановлено, можна в командній стрічці виконати:

```
pip –version
```

Отримаємо результат, схожий як на рисунку 8.19.

```
C:\Users\User>pip --version
pip 24.3.1 from C:\Users\User\AppData\Local\Programs\Python\Python313\Lib\site-packages\pip (python 3.13)
```

Рисунок 8.19 – Інформація про встановлену версію `pip`

Відсутність такого повідомлення може означати, що змінна `PATH` або неправильно вказує на розташування двійкових файлів Python, або не вказує на нього взагалі; наприклад, наша змінна PATH містить такий підрядок:

```
C:\Program Files\Python3\Scripts\;C:\Program Files\Python3\;
```

Найпростіший спосіб переналаштувати змінну `PATH` – перевстановити Python, надавши вказівку інсталятору встановити її за вас, інакше потрібно заходити в системні налаштування.

Щоб зрозуміти, що ще може `pip`, можна використати команду:

```
pip help
```

Отримаємо результат як на рисунку 8.20.

```
C:\Users\User>pip help

Usage:
  pip <command> [options]

Commands:
  install                     Install packages.
  download                    Download packages.
  uninstall                   Uninstall packages.
  freeze                      Output installed packages in requirements format.
  inspect                     Inspect the python environment.
  list                        List installed packages.
  show                        Show information about installed packages.
  check                       Verify installed packages have compatible dependencies.
  config                      Manage local and global configuration.
  search                      Search PyPI for packages.
  cache                       Inspect and manage pip's wheel cache.
  index                       Inspect information available from package indexes.
  wheel                       Build wheels from your requirements.
  hash                        Compute hashes of package archives.
  completion                  A helper command used for command completion.
  debug                       Show information useful for debugging.
  help                        Show help for commands.
```

Рисунок 8.20 – Довідка про `pip`

Якщо ми хочемо дізнатися більше про будь-яку з перелічених операцій, можемо використовувати наступну форму виклику `pip` :

```
pip help operation
```

Наприклад, рядок:

```
pip help install
```

покаже нам детальну інформацію про використання та налаштування параметрів команди `install`.

Популярні пакети наведено в таблиці 8.6.

Таблиця 8.6 – Популярні пакети Python

Галузь	Пакети
Дані/аналітика	`numpy, pandas, matplotlib`
HTTP-запити	`requests, httpx`
Веб-розробка	`flask, django, fastapi`
Машинне навчання	`scikit-learn, tensorflow, torch`
Автотестування	`pytest, unittest`

Контрольні питання

1. Що таке модуль у Python і яке його основне призначення?
2. Які переваги дає використання модулів у програмуванні?
3. Як створити власний модуль у Python?
4. Як імпортувати модуль у програму? Наведіть приклади різних способів імпортування.
5. У чому різниця між операторами import module, from module import name та import module as alias?
6. Що таке простір імен у Python і як він пов’язаний із модулями?
7. Для чого використовують псевдоніми під час імпорту модулів?
8. Як звернутися до функцій або змінних, визначених у модулі?
9. Які стандартні модулі Python ви знаєте? Наведіть приклади та коротко охарактеризуйте їх.
10. Що таке пакет (package) у Python і як він організований у файловій системі?
11. Яку роль відіграє файл `__init__.py` у структурі пакету?
12. Як відрізняється імпортування модулів від імпортування пакетів?
13. Що означає впорядкування програми за допомогою модулів і пакетів?
14. Що таке екосистема пакетів Python і яку роль у ній відіграє PyPI?
15. Як за допомогою інструменту `pip` виконати встановлення, оновлення чи видалення пакету на операційній системі Windows?

ТЕМА 9. Робота з текстовими даними

9.1 Сприйняття символів комп'ютером

Комп'ютери працюють лише з числами – точніше, з бітами (0 та 1). Тому для обробки символів використовується *кодування* – спосіб відповідності символу конкретному числовому значенню. Для перетворення символів на числа та навпаки використовуються стандартні таблиці. Найпоширеніші:

– ASCII – основна таблиця з 128 символів (таблиця 9.1);

– Extended ASCII – 256 символів;

– Unicode / UTF-8 – сучасний стандарт, підтримує тисячі символів усіх мов.

Таблиця 9.1 – Таблиця ASCII

Символ	Код	Символ	Код	Символ	Код	Символ	Код
(NUL)	0	(space)	32	@	64	`	96
(SOH)	1	!	33	A	65	a	97
(STX)	2	"	34	B	66	b	98
(ETX)	3	#	35	C	67	c	99
(EOT)	4	$	36	D	68	d	100
(ENQ)	5	%	37	E	69	e	101
(ACK)	6	&	38	F	70	f	102
(BEL)	7	'	39	G	71	g	103
(BS)	8	(	40	H	72	h	104
(HT)	9	)	41	I	73	i	105
(LF)	10	*	42	J	74	j	106
(VT)	11	+	43	K	75	k	107
(FF)	12	,	44	L	76	l	108
(CR)	13	-	45	M	77	m	109
(SO)	14	.	46	N	78	n	110
(SI)	15	/	47	O	79	o	111
(DLE)	16	0	48	P	80	p	112
(DC1)	17	1	49	Q	81	q	113
(DC2)	18	2	50	R	82	r	114
(DC3)	19	3	51	S	83	s	115
(DC4)	20	4	52	T	84	t	116
(NAK)	21	5	23	U	85	u	117
(SYN)	22	6	54	V	86	v	118
(ETB)	23	7	55	W	87	w	119
(CAN)	24	8	56	X	88	x	120
(EM)	25	9	57	Y	89	y	121
(SUB)	26	:	58	Z	90	z	122
(ESC)	27	;	59	[	91	{	123
(FS)	28	<	60	\	92	|	124
(GS)	29	=	61	]	93	}	125
(RS)	30	>	62	^	94	~	126
(US)	31	?	63	_	95		127

Код ASCII (скорочення від American Standard Code for Information Interchange) є найпоширенішим, майже всі сучасні пристрої (наприклад, комп'ютери, принтери, мобільні телефони, планшети тощо) використовують цей код. На основі кодів символів обчислюється їх двійкове значення. Приклад представлення деяких символів наведено в таблиці 9.2.

Таблиця 9.2 – Приклад бінарного кодування символів

Символ	Десятковий код	Бінарне подання
A	65	01000001
a	97	01100001
0	48	00110000
!	33	00100001

Мова Python містить функції, які дозволяють перетворювати код в символ і навпаки:

```
ord('A')  # дає 65 – код символу A
chr(65)   # дає 'A' – символ за його кодом
```

Розглянемо приклад (рисунок 9.1).

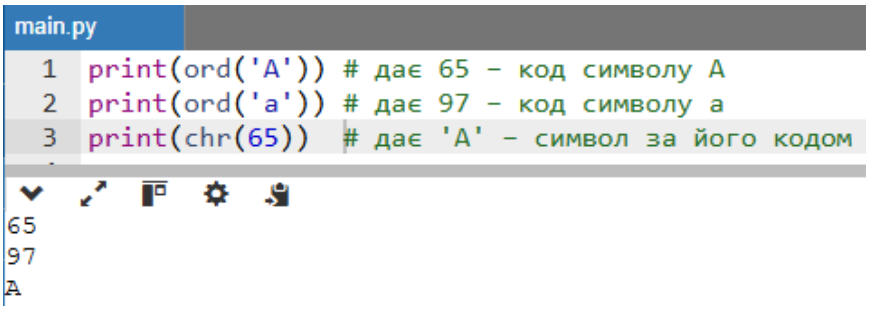


Рисунок 9.1 – Різниця для комп'ютера між великою та малою літерами

Як бачимо з рисунку, коди символів `A` та `a` різні, тому і їх бінарні коди будуть різні, відповідно з точки зору комп'ютера це абсолютно різні символи.

Звичайно, латинського алфавіту недостатньо для всього людства. Користувачі цього алфавіту в меншості. Треба було придумати щось більш гнучке і ємне, ніж ASCII, щось таке, що могло б зробити все програмне забезпечення в світі піддатливим до інтернаціоналізації, тому що різні мови використовують абсолютно різні алфавіти, і іноді ці алфавіти не такі прості, як латинський.

Класична форма коду ASCII використовує *вісім бітів* для кожного знака. Вісім бітів означають 256 різних символів. Перші 128 використовуються для стандартного латинського алфавіту (як великі, так і малі символи).

Кодова точка – це число, яке створює символ. Наприклад, 32 – це кодова точка, яка робить пробіл у кодуванні ASCII. Можна сказати, що стандартний код ASCII складається з 128 кодових точок. Оскільки стандартний ASCII займає 128 із 256 можливих кодових точок, ми можемо використовувати для національного алфавіту лише решту 128. Цього недостатньо для всіх можливих мов, але може бути достатньо для однієї мови або для невеликої групи схожих мов.

Ми можемо по-різному встановити верхню половину кодових точок для різних мов. Така концепція називається *кодовою сторінкою*.

Кодова сторінка є *стандартом для використання верхніх 128 кодових точок для зберігання певних національних символів*. Наприклад, існують різні кодові сторінки для Західної та Східної Європи, кириличного і грецького алфавітів, арабської та мови іврит тощо.

Це означає, що одна й та сама кодова точка може створювати різні символи при використанні на різних кодових сторінках.

Наприклад, кодова точка 200 перетворює Č (літера, яка використовується в деяких слов'янських мовах), коли використовується кодова сторінка ISO/IEC 8859-2, і перетворюється на Ш (літера кирилиці), коли використовується кодова сторінка ISO/IEC 8859-5.

Як наслідок, щоб визначити значення конкретної кодової точки, ми повинні знати цільову кодову сторінку.

Іншими словами, кодові точки, отримані з концепції кодової сторінки, неоднозначні.

Юнікод призначає унікальні (однозначні) символи (літери, дефіси, ідеограми тощо) більш ніж мільйону кодових точок. Unicode кодує всі можливі символи – від китайських ієрогліфів до смайликів. Найпоширеніше представлення – UTF-8, яке зберігає символи у вигляді 1-4 байтів.

Перші 128 кодових точок Unicode ідентичні ASCII, а перші 256 кодових точок Unicode ідентичні кодовій сторінці ISO/IEC 8859-1 (кодова сторінка, розроблена для західноєвропейських мов).

Стандарт Unicode нічого не говорить про те, як кодувати та зберігати символи в пам'яті та файлах. Він лише називає всі доступні символи та призначає їх площинам (групі символів подібного походження, застосування чи природи).

Існує більше ніж один стандарт, що описує методи, які використовуються для реалізації Unicode в реальних комп'ютерах і комп'ютерних системах зберігання. Найзагальнішим з них є *UCS-4*. Назва походить від універсального набору символів.

UCS-4 використовує 32 біти (чотири байти) для зберігання кожного символу, а код – це лише унікальний номер кодових точок Unicode. Файл, що містить текст у кодуванні UCS-4, може починатися з позначки порядку байтів (BOM – Byte Order Mark), непридатної для друку комбінації бітів, яка повідомляє про природу вмісту файлу. Це може знадобитися деяким утилітам.

Як бачите, UCS-4 є досить марнотратним стандартом – він збільшує розмір тексту в чотири рази порівняно зі стандартним ASCII. Існують інші форми кодування текстів Unicode.

Одним із найпоширеніших є *UTF-8*.

Назва походить від формату перетворення Юнікоду. UTF-8 використовує стільки бітів для кожної точки коду, скільки дійсно потрібно для їх представлення. Наприклад:

– усі латинські символи (і всі стандартні символи ASCII) займають вісім біт;

– нелатинські символи займають 16 біт;

– ідеографи CJK (Китай-Японія-Корея) займають 24 біти.

Через особливості методу, який використовується UTF-8 для зберігання кодових точок, немає необхідності використовувати специфікацію, але деякі інструменти шукають її під час читання файлу, і багато редакторів налаштовують її під час збереження.

Python 3 повністю підтримує Unicode та UTF-8.

Отже, комп’ютер перетворює символ в унікальний числовий код, а далі в двійкове число. Ці числа зберігаються та обробляються в пам’яті комп’ютера.

9.2 Рядки в Python

Рядок – це послідовність символів. Наприклад, «hello» – це рядок, що складається з набору символів: 'h', 'e', 'l', 'l' та 'o'. Рядки є *незмінними послідовностями*.

Розглянемо приклад, наведений на рисунку 9.2.

main.py

```
1  # Example 1
2  word = 'by'
3  print(len(word))
4
5  # Example 2
6  empty = ''
7  print(len(empty))
8
9  # Example 3
10 i_am = 'I\'m'
11 print(len(i_am))
12 print(i_am)
```

```
2
0
3
I'm
```

Рисунок 9.2 – Приклад знаходження довжини рядка

Функція `len()` повертає довжину рядка (кількість символів). Для прикладу `1` довжина фрази `by` становить `2` символи.

Будь-який рядок може бути порожнім. Його довжина становить `0` як у прикладі `2`.

Пригадаймо, що зворотна коса риска (\), яка використовується як екрануючий символ, не входить до загальної довжини рядка. Таким чином, код у прикладі `3` виводить `3`.

Для представлення рядка в Python можуть використовуватися подвійні чи одинарні лапки. Наприклад:

```
# Створення рядка за допомогою подвійних лапок
string1 = "Python programming"

# Створення рядка за допомогою одинарних лапок
string1 = 'Python programming'
```

Проте, якщо ми хочемо, щоб рядок займав кілька рядків, то його варто позначити з обох сторін потрійними лапками (три апострофи).

Розглянемо приклад, наведений на рисунку 9.3.

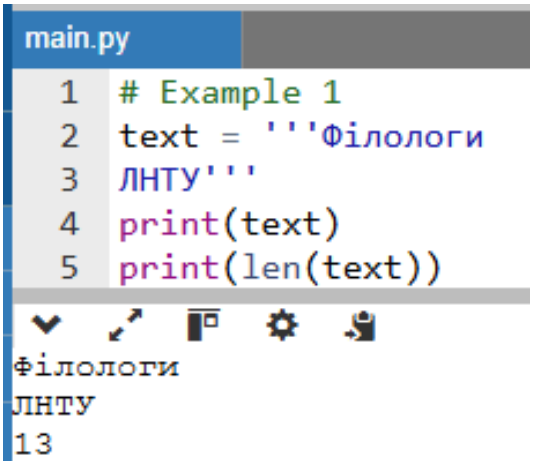


Рисунок 9.3 – Багатострічковий рядок

Якщо уважно подивитися на фразу з рисунка 9.3, то можна порахувати, що слово «філологи» складається з 8 символів, «ЛНТУ» – з чотирьох, разом 12 символів, проте програма нам видала результат – 13 символів. Вся справа в тому, що ми натискали enter після «філологи» – це невидимий символ переходу на новий рядок (в Python це \n).

Багаторядкові рядки також можуть бути розділені потрійними лапками:

```
multiline = """Line #1
Line #2"""
```

Отже, рядок – це послідовність символів, взятих у лапки.

Можна використовувати:

– одинарні лапки 'текст';

– подвійні лапки "текст";

– потрійні лапки '"текст"' або """текст""" (для багаторядкових рядків).

9.3 Операції над рядками

Як і інші типи даних, рядки мають власний набір дозволених операцій, хоча вони досить обмежені порівняно з числами.

Загалом рядки можуть бути:

– об'єднані (з'єднані) – за допомогою оператора **+**;

– тиражовані – за допомогою оператора *.

Звернімо увагу: оператор **+** для рядків це не додавання, оператор * для рядків це не множення. Можливість використовувати той самий оператор для абсолютно різних типів даних (наприклад, чисел та рядків) називається *перевантаженням* (оскільки такий оператор перевантажений різними обов'язками). Приклад використання операторів з'єднання та тиражування наведено на рисунку 9.4.

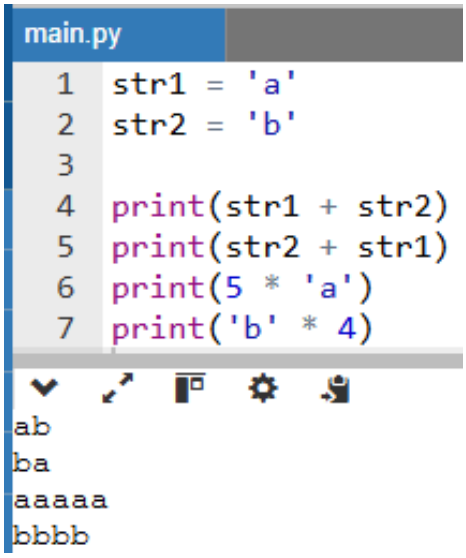


Рисунок 9.4 – З'єднання та тиражування рядків

Оператор **+** використовується для двох або більше рядків, створює новий рядок, що містить усі символи з його аргументів (примітка: порядок має значення

перевантажений + для рядків, на відміну від свого числового варіанту, не є комутативним).

Оператору * потрібні рядок і число як аргументи; у цьому випадку порядок не має значення – ми можемо поставити число перед рядком, або навпаки, результат буде той самий – новий рядок, створений n-ою копією рядка аргументу.

Якщо ми хочемо дізнатися значення коду ASCII/UNICODE певного символу, можемо скористатися функцією під назвою `ord()`. Функції потрібен односимвольний рядок як аргумент – порушення цієї вимоги спричиняє виняток `TypeError` (рисунок 9.5).

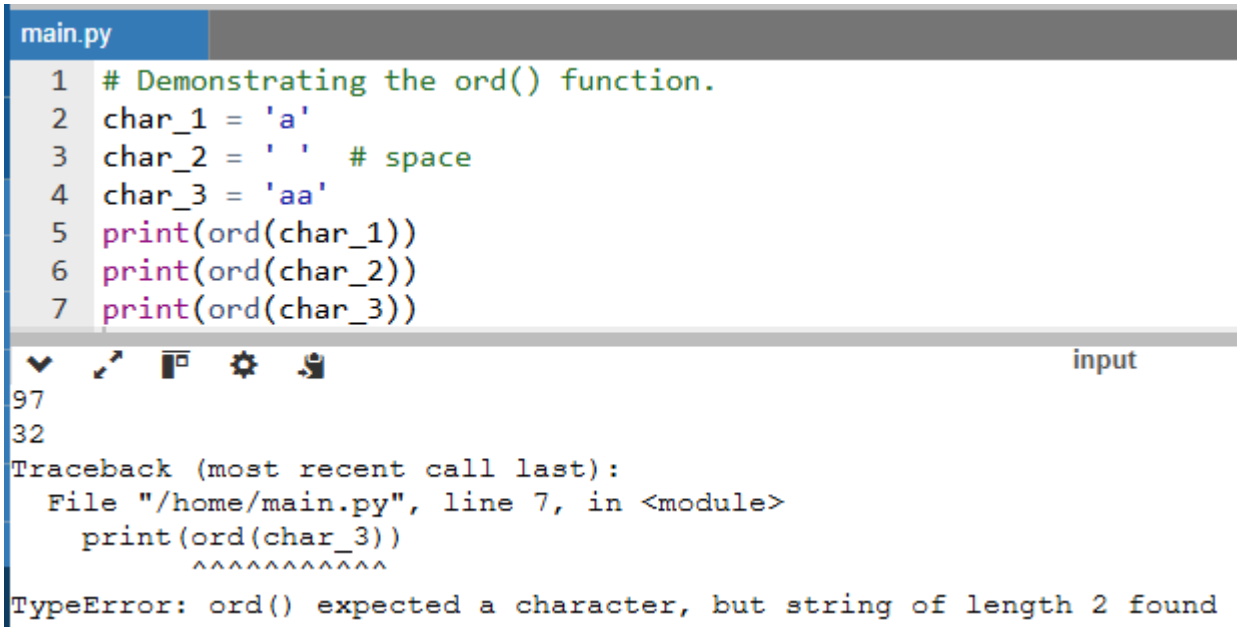


Рисунок 9.5 – Приклад використання функції `ord()`

Якщо ми знаємо кодову точку (число) і хочемо отримати відповідний символ, ми можемо скористатися функцією з назвою `chr()`. Функція приймає кодову точку та повертає її символ. Виклик його з недійсним аргументом (наприклад, негативний або недійсний код) викликає винятки `ValueError` або `TypeError` .

9.4 Рядки як послідовності

Рядки не є списками, але ми можемо розглядати їх як списки в багатьох конкретних випадках. Наприклад, якщо ми хочемо отримати доступ до будь-якого символу рядка, можемо зробити це за допомогою індексування (рисунок 9.6).

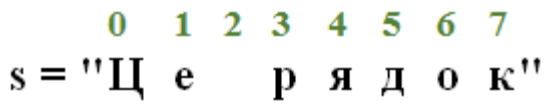


Рисунок 9.6 – Рядок як послідовність

Доступ до символів рядка в Python може здійснюватися трьома способами:

– індексація – розгляд рядка як списку та отримання доступу до символу за його індексом. Спроба перейти за межі рядка спричинить помилку. Наприклад:

```
s = 'Це рядок'
print(s[3])  # р
print(s[15]) #помилка – вихід за межі стрічки
```

– від'ємна індексація – подібно до списку, в Python дозволена від'ємна індексація для доступу до символів рядка. Наприклад:

```
s = 'Це рядок'
print(s[-3])  # д
```

– зріз – отримання доступу до діапазону символів рядка за допомогою оператора зрізу `:`. Приклад наведено на рисунках 9.7-9.8.

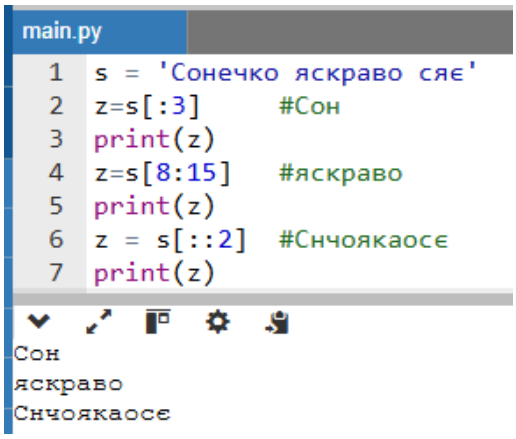


Рисунок 9.7 – Приклади зрізів у рядку

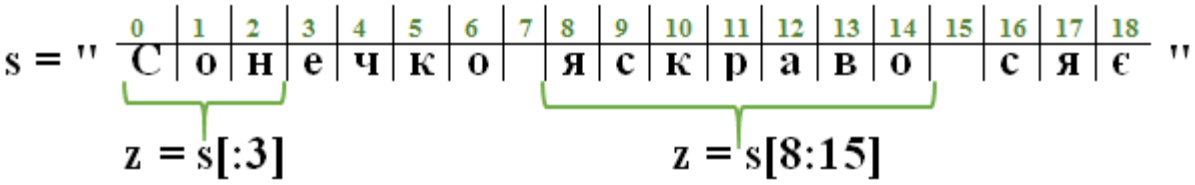


Рисунок 9.8 – Ілюстрація зрізів у рядку

Оператор добування зрізу з рядка: `z = s[i : j : step]`, де `z` – змінна, в яку зберігається підрядок рядка `s`; `i` – індекс початку зрізу; `j` – обмеження справа (не входить у зріз); `step` – з яким кроком вибираються символи.

Також для роботи з рядками можна застосовувати оператор циклу (рисунок 9.9).

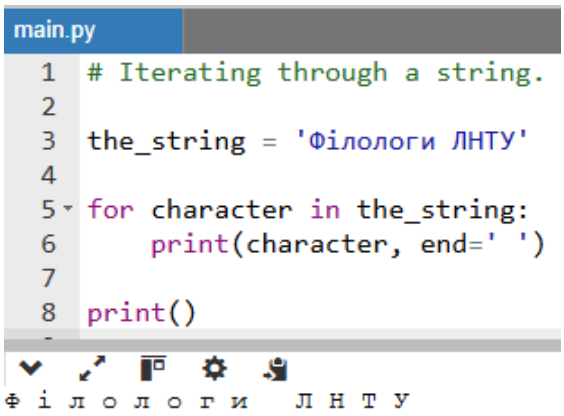


Рисунок 9.9 – Ітераційний доступ до символів рядка за допомогою циклу

Рядки в Python є *іммутабельними*. Це означає, що їх символи не можуть бути змінені. Наприклад:

```
message = 'Hola Amigos'
message[0] = 'H'
print(message)
```

Результат:

```
TypeError: 'str' object does not support item assignment
```

Однак, змінній можна присвоїти нове рядкове значення. Наприклад:

```
message = 'Hola Amigos'
```

```
# Присвоюємо змінній message нове значення
message = 'Hello Friends'
print(message) # виведе "Hello Friends"
```

Для порівняння двох рядків використовується оператор `==`. Якщо рядки однакові, оператор поверне `True`, в протилежному випадку – `False`. Наприклад:

```
str1 = "Hello, world!"
str2 = "I love Python."
str3 = "Hello, world!"

# Порівняння рядків str1 та str2
print(str1 == str2)

# Порівняння рядків str1 та str3
print(str1 == str3)
```

Результат:

```
False
True
```

У цьому прикладі `str1` та `str2` не є однаковими, отже, результат `False`; `str1` та `str3` є однаковими, отже, результат `True`.

Так як рядки – незмінні послідовності, то з рядками не можна так робити:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
del alphabet[0]    #Помилка
alphabet.append("A")    #Помилка
alphabet.insert(0, "A")     #Помилка
```

Але рядки можна перезаписувати (рисунок 9.10).

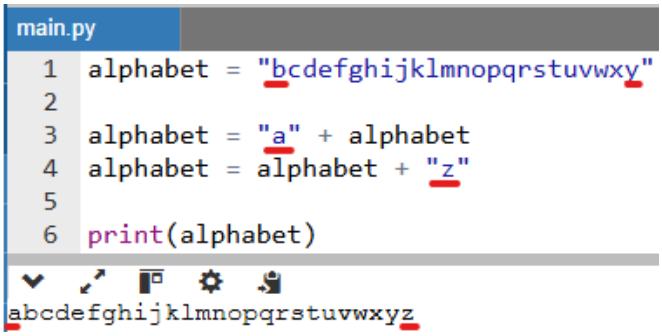


Рисунок 9.10 – Перезапис рядка

9.5 Оператори in та not in та інші функції опрацювання рядків

Оператори `in` та `not in` використовуються для *перевірки наявності* елемента (символа, підрядка, числа тощо) в рядку.

`in` та `not in` повертають логічне значення: `True` або `False`, часто використовуються в умовах (`if`, `while`) або для фільтрації даних.

Оператор `in` перевіряє, чи елемент (символ або інший рядок) є усередині рядка. Наприклад:

```
print('a' in 'apple')  # True
print('app' in 'apple')  # True
print('z' in 'apple')  # False
```

Оператор `not in` перевіряє, що елемента немає у послідовності. Наприклад:

```
print('z' not in 'apple')  # True
```

Функція `min()` знаходить мінімальний елемент рядка. Є одна умова – рядок не може бути порожнім, інакше отримаємо виняток `ValueError`. Розглянемо приклад, наведений на рисунку 9.11.

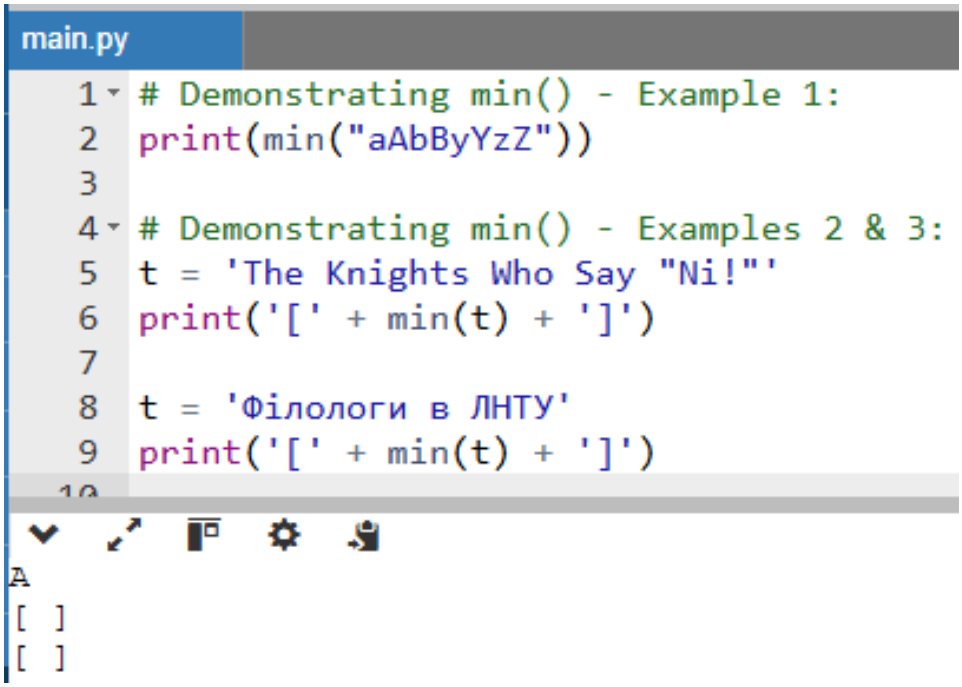


Рисунок 9.11 – Приклад застосування функції `min()`

Другий рядок цього коду виведе літеру А, адже її код в таблиці ASCII найменший, шостий та дев'ятий рядки коду виведуть пробіли, бо код пробілу 32 – це менше за код будь-якої літери англійського або українського алфавіту.

Подібним чином функція `max()` знаходить максимальний елемент послідовності.

Функція `list()` приймає як аргумент рядок і створює новий список, що містить усі символи рядка, по одному на елемент списку (рисунок 9.12). Ця функція також може створювати новий список з багатьох інших сутностей (наприклад, з кортежів і словників).

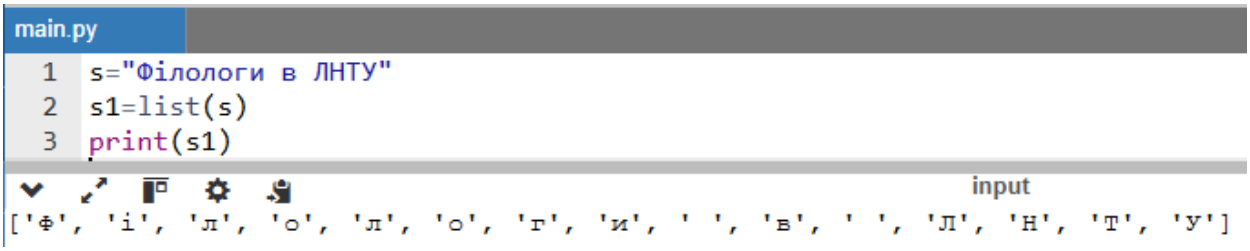


Рисунок 9.12 – Приклад застосування функції `list()`

9.6 Методи роботи з рядками

Функції для рядків *приймають рядок* як аргумент:

```
функція(рядок)
```

Методи для роботи з рядками прив'язані до об'єкта (рядка), вони викликаються через крапку:

```
рядок.метод()
```

9.6.1 Метод `index()`

Метод `index()` шукає послідовність з початку, щоб знайти перший елемент значення, указаного в його аргументі. Примітка: шуканий елемент має бути в послідовності – його відсутність спричинить виняток `ValueError` (рисунок 9.13). Цей метод повертає індекс першого входження аргументу (це

означає, що найменший можливий результат дорівнює `0`, а найбільший – це довжина аргументу, зменшена на `1`).

```
main.py
1 # Demonstrating the index() method:
2 print("aAbByYzZaA".index("b"))
3 print("aAbByYzZaA".index("Z"))
4 print("aAbByYzZaA".index("A"))
5 print("aAbByYzZaA".index("r"))
2
7
1
Traceback (most recent call last):
  File "/home/main.py", line 5, in <module>
    print("aAbByYzZaA".index("r"))
          ^^^^^^^^^^^^^^^^^^^^^^^
ValueError: substring not found
```

Рисунок 9.13 – Приклад застосування методу `index()`

9.6.2 Метод `count()`

Метод `count()` підраховує всі входження елемента в послідовність. Відсутність таких елементів не викликає проблем (рисунок 9.14).

```
main.py
1 # Demonstrating the count() method:
2 print("ЛНТУ".count("У"))
3 print('ЛНТУ'.count("А"))
1
0
```

Рисунок 9.14 – Приклад застосування методу `count()`

9.6.3 Метод `capitalize()`

Метод `capitalize()` створює новий рядок, заповнений символами, взятими з вихідного рядка, але намагається змінити їх у такий спосіб: якщо перший символ (з індексом `0`) у рядку є літерою, він буде перетворений у верхній

регістр, всі інші літери рядка, що залишилися, будуть перетворені на малі (рисунок 9.15).

```
main.py
1 # Demonstrating the capitalize() method:
2 print('aBcD'.capitalize())
3 print('ми Вчимося в ЛНТУ'.capitalize())
4 print('1 студент'.capitalize())
```

```
Abcd
Ми вчимося в лнту
1 студент
```

Рисунок 9.15 – Приклад застосування методу `capitalize()`

9.6.4 Метод `center()`

Метод `center()` створює копію вихідного рядка, намагаючись розташувати його всередині поля заданої ширини, тобто він виконує центрування шляхом додаванням пробілів перед і після рядка (рисунок 9.16).

Двопараметричний варіант `center()` використовує символ із другого аргументу замість пробілу.

```
main.py
1 # Demonstrating the center() method:
2 print('[' + 'ЛНТУ'.center(10) + ']')
3 print('[' + 'ЛНТУ'.center(10, '-') + ']')
```

```
[   ЛНТУ   ]
[---ЛНТУ---]
```

Рисунок 9.16 – Приклад застосування методу `center()`

9.6.5 Методи `endswith()` *та* `startswith()`

Метод `endswith()` перевіряє, чи даний рядок закінчується вказаним аргументом, і повертає `True` або `False` залежно від результату перевірки (рисунок 9.17).

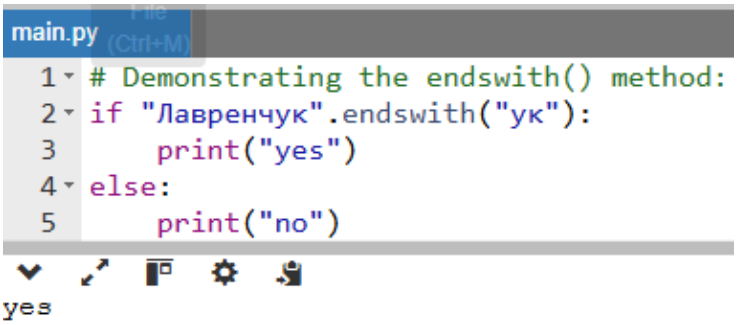


Рисунок 9.17 – Приклад застосування методу `endswith()`

Метод є `startswith()` дзеркальним відображенням `endswith()` – він перевіряє, чи даний рядок починається з вказаного підрядка.

9.6.6 Методи `find()` та `rfind()`

Метод `find()` схожий на метод `index()` – він шукає підрядок і повертає індекс першого входження цього підрядка, але він не генерує помилку для аргументу, що містить неіснуючий підрядок, натомість він повертає -1 (рисунок 9.18). На відміну від методу `index()`, що застосовується для різних послідовностей, метод `find()` працює лише з рядками. Не варто використовувати `find()` якщо ми хочемо перевірити, чи зустрічається лише один символ у рядку – оператор `in` буде значно швидшим.

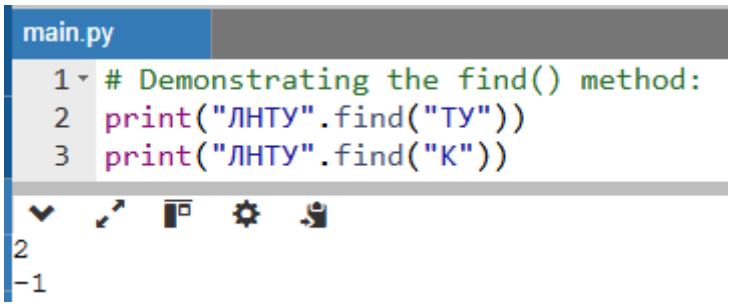


Рисунок 9.18 – Приклад застосування методу `find()`

Якщо ми хочемо виконати пошук не з початку рядка, а з будь-якої позиції, можемо використовувати двопараметричний варіант методу `find()`. Наприклад:

```
print('kappa'.find('a', 2))      # 4
```

Другий аргумент визначає індекс, з якого буде розпочато пошук (він не обов'язково повинен поміщатися в рядок). Серед двох букв `a` знайдеться лише друга.

Ми можемо використовувати метод `find()` для пошуку всіх входжень підрядка (рисунок 9.19).

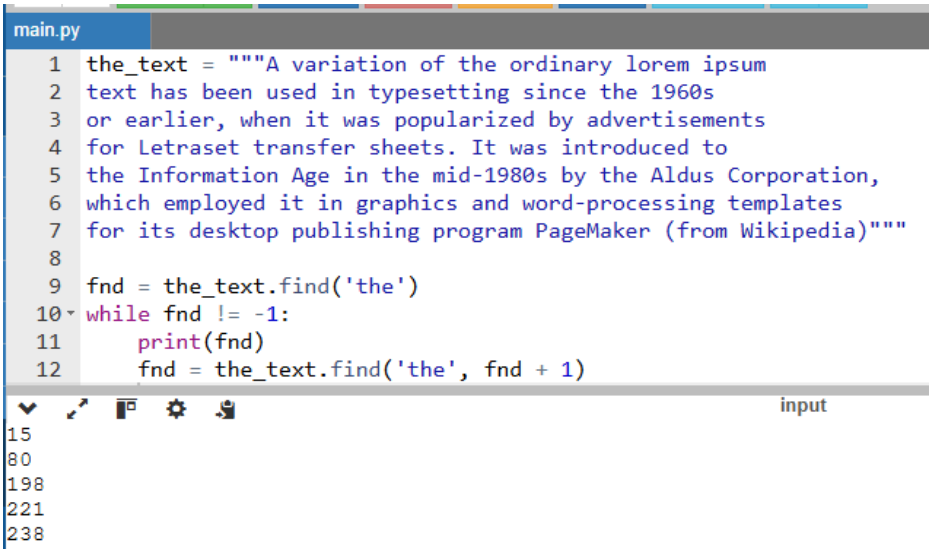


Рисунок 9.19 – Приклад знаходження всіх позицій артикля `the`

Існує також трипараметричний варіант методу `find()` – третій аргумент вказує на перший індекс, який не буде враховано під час пошуку (фактично це верхня межа пошуку):

```
print('kappa'.find('a', 1, 4))  # 1
print('kappa'.find('a', 2, 4))  # -1
```

Методи `rfind()` з одним, двома чи трьома параметрами виконують майже те ж саме, що й їх відповідники (позбавлені префікса `r`), але починають пошук із кінця рядка, а не з початку (префікс `r` від слова `right`).

9.6.7 Методи isalnum(), isalpha(), isdigit(), islower(), isupper() та isspace()

Метод `isalnum()` перевіряє, чи рядок містить лише цифри або букви (літери), і повертає `True` або `False` відповідно до результату (рисунок 9.20). будь-який елемент рядка, який не є цифрою чи літерою, спричиняє повернення методом `False`. Порожній рядок також.

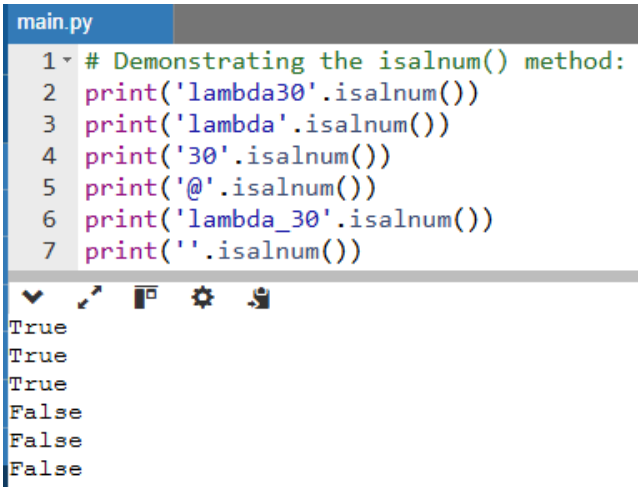


Рисунок 9.20 – Приклад застосування методу `isalnum()`

Метод `isalpha()` працює подібним чином, але його цікавлять літери:

```
# Example 1: Demonstrating the isapha() method:
print("ЛНТУ".isalpha())     # True
print('ЛНТУ5'.isalpha())    # False
```

Метод `isdigit()` розглядає лише цифри – все інше створює результат `False`.

Метод `islower()` є частковим варіантом `isalpha()` – він сприймає лише малі літери, все решта для нього `False`.

Метод `isupper()` є частковим варіантом `isalpha()` – він сприймає лише великі літери, все решта для нього `False`.

Метод `isspace()` ідентифікує лише пробіли – він ігнорує будь-які інші символи (результат `False`).

9.6.8 Метод `join()`

Метод `join()` виконує об'єднання, він очікує один аргумент у вигляді списку (нагадаємо, що список береться в квадратні дужки `[]`), при цьому необхідно переконатися, що всі елементи списку є рядками – інакше метод викличе виняток `TypeError` (рисунок 9.21). В результаті виконання цього методу усі елементи списку будуть об'єднані в один рядок, але рядок, з якого було викликано метод, використовується як роздільник , розміщений серед рядків списку, у результаті повертається новостворений рядок.

```
main.py
1 # Demonstrating the join() method:
2 print(",".join(["ЛНТУ", "ПЛ", "2025"]))
3 print(",".join(["ЛНТУ", "ПЛ", 2025]))
                                                    input
ЛНТУ,ПЛ,2025
Traceback (most recent call last):
  File "/home/main.py", line 3, in <module>
    print(",".join(["ЛНТУ", "ПЛ", 2025]))
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: sequence item 2: expected str instance, int found
```

Рисунок 9.21 – Приклад застосування методу `join()`

9.6.9 Методи `lower()`, `upper()`, `swapcase()` та `title()`

Метод `lower()` не приймає жодних параметрів, він робить копію вихідного рядка, замінює всі літери у верхньому регістрі на відповідники в нижньому регістрі та повертає рядок як результат. Знову ж таки, вихідний рядок залишається недоторканим. Якщо рядок не містить символів верхнього регістру, метод повертає оригінальний рядок (рисунок 9.22).

Метод `upper()` робить копію вихідного рядка, замінює всі літери нижнього регістру на відповідники у верхньому регістрі та повертає рядок як результат (рисунок 9.22).

```
main.py
1 # Demonstrating the lower() method:
2 print("SiGmA=60".lower())
3 # Demonstrating the upper() method:
4 print("I know that I know nothing. Part 2.".upper())
sigma=60
I KNOW THAT I KNOW NOTHING. PART 2.
```

Рисунок 9.22 – Приклад застосування методів `lower()` та `upper()`

Метод `swapcase()` створює новий рядок, змінюючи регістри всіх літер у вихідному рядку: символи нижнього регістру стають великими, і навпаки. Усі інші символи (не літери) залишаються недоторканими (рисунок 9.23).

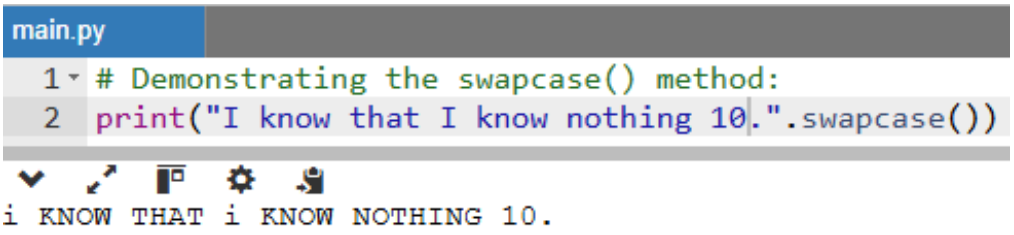


Рисунок 9.23 – Приклад застосування методу `swapcase()`

Метод `title()` виконує подібну функцію – він змінює першу літеру кожного слова на прописну, а всі інші перетворює на малі (рисунок 9.24).

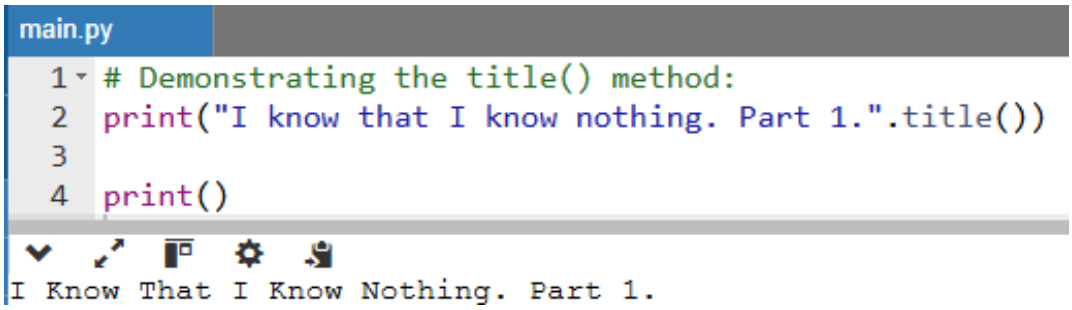


Рисунок 9.24 – Приклад застосування методу `title()`

9.6.10 Методи `lstrip()`, `rstrip()` *та* `strip()`

Версія методу `lstrip()` без параметрів повертає щойно створений рядок, утворений з оригінального шляхом видалення всіх пробілів на початку:

```
print("[" + " tau ".lstrip() + "]")  #[tau ]
```

Тут дужки не є частиною результату – вони лише показують межі результату.

Версія методу `lstrip()` з одним параметром робить те саме, що й його версія без параметрів, але видаляє не пробіли, а всі символи, які передано як аргумент методу:

```
print("www.lntu.edu.ua".lstrip("w.")) #lntu.edu.ua
```

Два варіанти методу `rstrip()` роблять майже те саме, що `lstrips()`, але впливають на протилежну сторону рядка, тобто зміни відбуваються з кінця рядка:

```
# Demonstrating the rstrip() method:
print("[" + " lntu ".rstrip() + "]")    #[ lntu]
print("lntu.edu.ua".rstrip("ua"))       #lntu.edu.
```

Метод `strip()` поєднує в собі ефекти, викликані `rstrip()` і `lstrip()` – створює новий рядок без усіх пробілів на початку та в кінці:

```
print("[" + "   Ми в ЛНТУ    ".strip() + "]")#[Ми в ЛНТУ]
```

9.6.11 Метод `replace()`

Метод `replace()` з двома параметрами повертає копію оригінального рядка, у якій усі випадки першого аргументу замінено другим аргументом (рисунок 9.25).

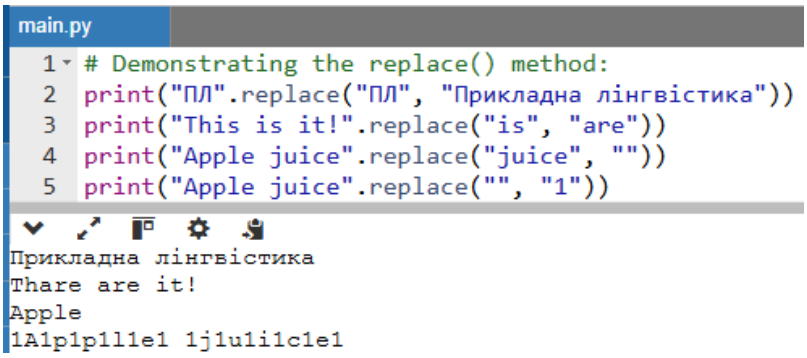


Рисунок 9.25 – Приклад застосування `replace()` з двома параметрами

Як бачимо з рисунка, якщо другий аргумент є порожнім рядком, заміна фактично означає видалення частини рядка першого аргументу, якщо ж перший аргумент порожній рядок, то його символи будуть розділені другим аргументом.

Метод `replace()` з трьома параметрами використовує третій аргумент (число), щоб обмежити кількість замін (рисунок 9.26).

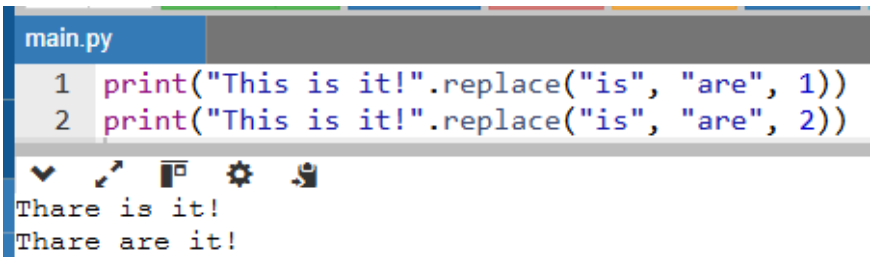


Рисунок 9.26 – Приклад застосування `replace()` з трьома параметрами

10.6.12 Метод `split()`

Метод `split()` розділяє рядок на список усіх виявлених підрядків. Метод передбачає, що підрядки розділені пробілами – пробіли не беруть участі в операції і не копіюються в результуючий список (рисунок 9.27). Якщо рядок порожній, результуючий список також буде порожнім.

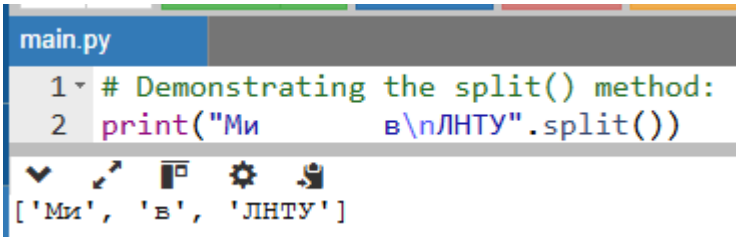


Рисунок 9.27 – Приклад застосування `split()`

Протилежну дію можна виконати за допомогою методу `join()`.

9.7 Сортування та інша обробка текстових даних

Рядки Python можна порівнювати за допомогою того самого набору операторів, які використовуються по відношенню до чисел (= =, !=, >, >=, <, <=). При цьому варто пам’ятати, що Python не розуміє тонких лінгвістичних проблем – він просто порівнює значення кодових точок символ за символом.

Два рядки є рівними, якщо вони складаються з однакових символів у однаковому порядку. Таким же чином два рядки не є рівними, якщо вони не складаються з однакових символів у тому самому порядку:

```
'alpha' == 'alpha'  # True
'alpha' != 'Alpha'  # True
```

Остаточний результат порівняння визначається шляхом *порівняння першого різного символу в обох рядках* (пам’ятаймо про коди ASCII/UNICODE).

Коли ми порівнюємо два рядки різної довжини, і коротший з них ідентичний початку довшого, то довший рядок вважається більшим:

```
'alpha' < 'alphabet' # True
```

Порівняння рядків завжди чутливе до регістру (*великі літери вважаються меншими, ніж малі*).

```
'beta' > 'Beta'# True
```

Навіть якщо рядок містить лише цифри, це все одно не число (рисунок 9.28).

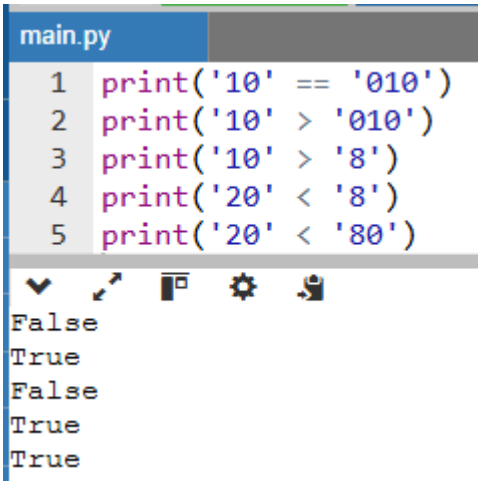


Рисунок 9.28 – Приклад порівняння рядків

Сортування є складнішим випадком порівняння.

Загалом, Python пропонує два різні способи сортування списків.

Перший реалізований як функція з іменем `sorted()`. Функція приймає один аргумент (список) і повертає новий список, заповнений елементами відсортованого аргументу. Оригінальний список залишається незмінним (рисунок 9.29).

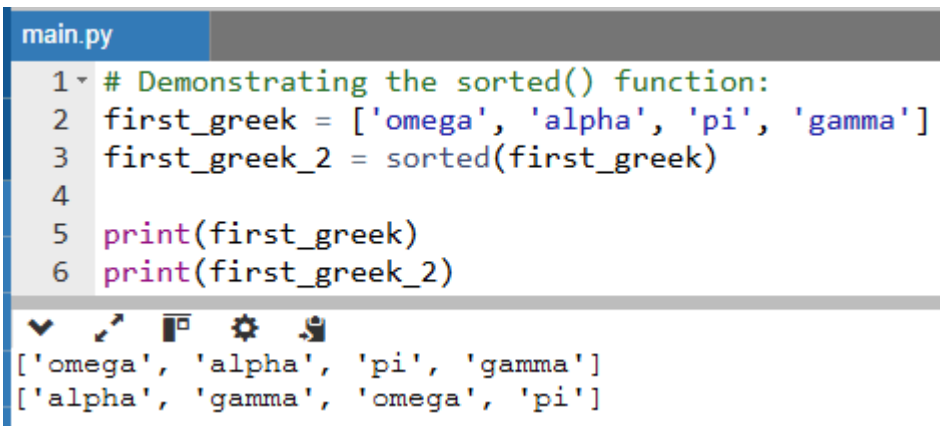


Рисунок 9.29 – Приклад використання функції `sorted()`

Другий спосіб впливає на сам список – *новий список не створюється* (рисунок 9.30). Упорядкування виконується на місці методом `sort()`.

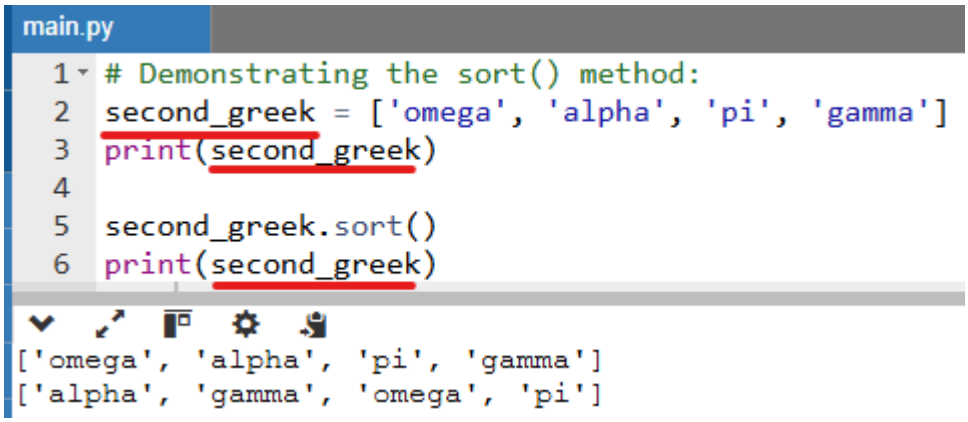


Рисунок 9.30 – Приклад використання методу `sort()`

Функцією під назвою `str()` ми можемо перетворити число (ціле число або число з плаваючою точкою) у рядок (рисунок 9.31).

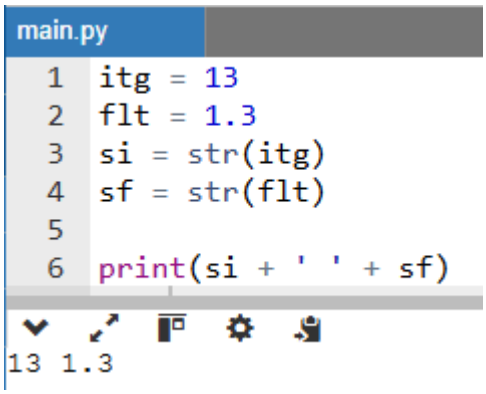


Рисунок 9.31 – Приклад перетворення числа в рядок

Зворотне перетворення (рядок-число) можливе тоді і тільки тоді, коли рядок представляє число за допомогою функції `int()` для цілих чисел або `float()` – для дійсних.

Контрольні питання

1. Як комп’ютер сприймає символи та текстові дані? Що таке кодування символів?
2. Які основні кодування тексту використовуються у Python?
3. Що таке рядок у Python і як його створити?

4. Чим відрізняється створення рядків у одинарних, подвійних і потрійних лапках?

5. Як у Python виконуються основні операції над рядками?

6. Що означає, що рядки в Python є послідовностями?

7. Як отримати доступ до окремих символів рядка за допомогою індексації та зрізів?

8. Для чого використовуються оператори `in` та `not in` при роботі з рядками?

9. Як перевірити, чи певний підрядок входить до рядка? Наведіть приклад.

10. Назвіть основні методи роботи з рядками.

11. Як можна розділити рядок на окремі слова або символи?

12. Як виконати з’єднання елементів списку в рядок за допомогою методу `join()`?

13. Як у Python відбувається сортування текстових даних?

14. Яким чином можна перетворити символи на коди (числа) та навпаки?

ТЕМА 10. Робота з файлами

10.1 Доступ до файлів із коду Python

Файл – це контейнер для зберігання даних. Коли ми хочемо читати з файлу або записувати в нього, нам потрібно спочатку його відкрити. Після того, як ми закінчили читання/запис, нам потрібно закрити файл, щоб звільнити ресурси, пов'язані з ним.

Таким чином, в Python операції з файлами виконуються в наступному порядку:

– відкрити файл;

– читання або запис;

– закрити файл.

У Python для роботи з файлами є вбудовані функції та методи, що дозволяють:

– відкривати файли;

– читати їхній вміст;

– записувати дані;

– закривати файли.

10.1.1. Відкриття файлу

Для відкриття файлу використовується функція `open()`.

Синтаксис:

```
file = open('ім'я_файлу', 'режим')
```

Поширені режими роботи:

– `'r'` – читання (default);

– `'w'` – запис (перезаписує файл);

– `'a'` – дозапис у кінець файлу;

– `'b'` – робота з бінарними файлами;

– `'+'` – читання і запис.

Основні методи для файлів:

– `read()` – читає весь файл;

– `readline()` – читає один рядок;

– `readlines()` – читає всі рядки у список;

– `write(text)` – записує текст;

– `writelines(list)` – записує список рядків;

– `close()` – закриває файл.

10.1.2 Читання з файлу

Читати весь файл:

```
file = open('example.txt', 'r')
content = file.read()
print(content)
file.close()
```

Читати рядок за рядком:

```
file = open('example.txt', 'r')
line = file.readline()
print(line)
file.close()
```

Читати всі рядки у список:

```
file = open('example.txt', 'r')
lines = file.readlines()
print(lines)
file.close()
```

10.1.3 Запис у файл

Перезапис файлу:

```
file = open('example.txt', 'w')
file.write('Hello, world!')
file.close()
```

Дозапис у файл:

```
file = open('example.txt', 'a')
file.write('\nAnother line')
file.close()
```

10.1.4 Контекстний менеджер (`with`*)*

Рекомендований спосіб роботи з файлами – через `with`. Він автоматично закриває файл після завершення роботи.

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

10.1.5 Копіювання файлу

```
# Копіювання текстового файлу
with open('input.txt', 'r', encoding='utf-8') as source_file:
    with    open('copy.txt',    'w',    encoding='utf-8')    as
target_file:
        for line in source_file:
            target_file.write(line)
```

Алгоритм

– відкриваємо оригінальний файл `input.txt` у режимі читання (`'r'`);

– відкривається новий файл `copy.txt` у режимі запису (`'w'`);

– кожен рядок читається із `source_file` і одразу записується у `target_file`;

– обидва файли автоматично закриваються після завершення роботи завдяки `with`.

10.1.5 Обробка помилок

Щоб уникнути аварійного завершення програми, можна обгортати роботу з файлами у `try-except`:

```
try:
    with open('example.txt', 'r') as file:
        data = file.read()
except FileNotFoundError:
    print('Файл не знайдено!')
```

10.2 Імена файлів

При роботі з файлами важливо правильно вказувати імена файлів і розуміти деякі правила:

– *ім'я файлу* – це текстовий рядок, який вказує шлях до файлу на диску;

– *розширення файлу* часто визначає його тип (наприклад, .txt, .json, .py);

– ім'я може бути *відносним* або *абсолютним* шляхом (таблиця 10.1).

Таблиця 10.1 – Види шляхів до файлу

Тип шляху	Приклад	Пояснення
Відносний	`'data/file.txt'`	Відносно поточної папки
Абсолютний	`'C:/Users/User/Desktop/file.txt'`	Повний шлях від кореня файлової системи

Правила для імен файлів:

– імена чутливі до регістру на Linux та macOS (File.txt ≠ file.txt);

– не використовуйте заборонені символи, наприклад для Windows: \ / : * ? " < > |;

– уникайте пробілів та спеціальних символів у іменах, краще замінювати їх на _ або -;

– використовуйте розширення файлу для вказівки його формату (наприклад, `.txt` для тексту).

Розглянемо приклади відкриття файлів з різними іменами:

```
# Відносний шлях
with open('notes.txt', 'r') as f:
    content = f.read()

# Абсолютний шлях (Windows)
with open('C:/Users/Anna/Documents/notes.txt', 'r') as f:
    content = f.read()
# Шлях до файлу в папці
with open('data/records.csv', 'r') as f:
    data = f.read()
```

10.3 Потоки

Файловий об'єкт (`file object`) або *потік* (`stream`) – об'єкт, який представляє файл-орієнтований програмний інтерфейс (методи `read()`, `write()` та інші) для доступу до певного ресурсу.

Залежно від способу створення, файловий об'єкт може надавати доступ до реального файлу на диску або іншого виду пристрою зберігання чи передачі даних (стандартні потоки вводу/виводу, буфери у пам'яті, сокети, інше).

Як правило операційна система створює три потоки, які називають стандартними. При запуску процесу операційна система надає процесу доступ до цих потоків.

Стандартними потоками є:

– `sys.stdin` – за замовчуванням пов'язаний з клавіатурою терміналу, потік стандартного введення;

– `sys.stdout` – за замовчуванням пов'язаний з екраном термінала, потік стандартного виведення;

– `sys.stderr` – за замовчуванням пов'язаний з екраном термінала, потік для повідомлень про помилки.

Приклад:

```
import sys

name = sys.stdin.readline()  # зчитуємо введення
sys.stdout.write(f'Hello, {name}\n')# виводимо привітання
```

Файл у Python – це теж потік, відкриття файлу створює потік для читання або запису.

Файли в Python поділяються на бінарні та текстові.

У свою чергу бінарні файли поділяються на буферизовані та небуферизовані.

Текстові файли вважаються буферизованими, тобто дані накопичуються у пам'яті перед передачею.

10.4 Ієрархія класів та робота з файлами

У Python робота з потоками введення/виведення (`I/O streams`) реалізується через класову ієрархію, яка дозволяє працювати з різними типами потоків – текстовими, бінарними, файлами, пам'яттю тощо.

Уся ця ієрархія базується на модулі `io`.

Модуль `io` забезпечує засоби для роботи з різними типами вводу/виводу. У мові Python забезпечуються 3 основні типи вводу/виводу:

– текстовий ввід/вивід – забезпечує роботу з рядковими об’єктами типу `str`;

– двійковий або бінарний ввід/вивід – забезпечує роботу з двійковими об’єктами типу `bytes`;

– буквальний ввід/вивід (`raw input/output`) або небуферизований ввід/вивід.

Для кожного з типів вводу/виводу створюється файловий об’єкт, який ще називають потоковий об’єкт.

На рисунку 10.1 зображено ієрархію класів вводу/виводу.

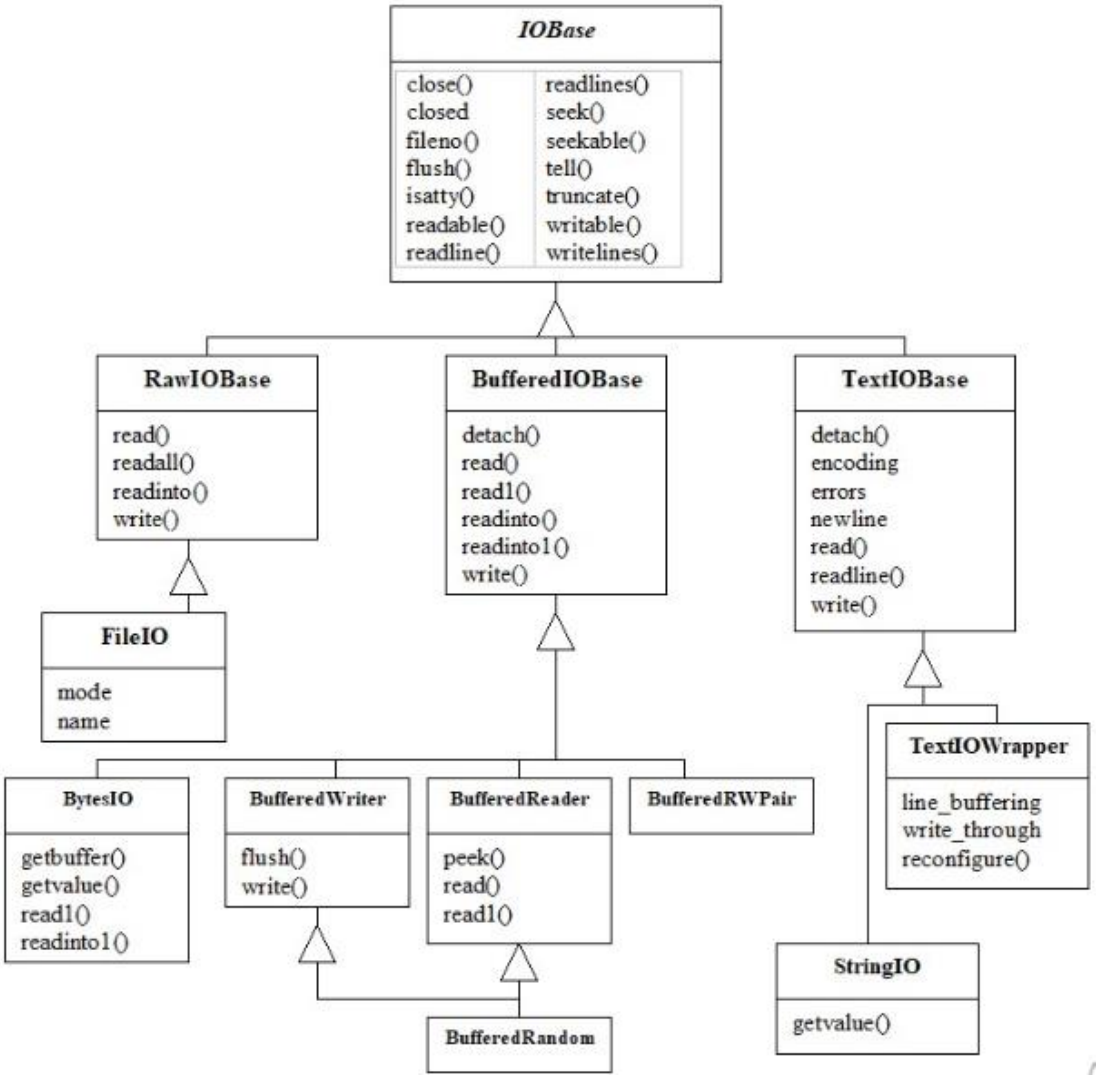


Рисунок 10.1 – Ієрархія класів вводу/виводу

Ієрархія класів вводу/виводу:

– `IOBase` – базовий абстрактний клас, вершина ієрархії;

– `RawIOBase` – абстрактний клас небуферизованих бінарних файлів;

– `FileIO` – клас для роботи з небуферизованими бінарними потоками;

– `BufferedIOBase` – абстрактний клас буферизованих бінарних файлів;

– `BufferedReader` – зчитування даних з буферизованих бінарних потоків;

– `BufferedWriter` – запис даних у буферизовані бінарні потоки;

– `BufferedRWPair` – зчитування і запис даних для буферизованих бінарних потоків;

– `BytesIO` – «віртуальний» потік (як правило у оперативній пам'яті);

– `TextIOBase` – абстрактний клас текстових файлів;

– `StringIO` – «віртуальний» потік символів Unicode;

– `TextIOWrapper` – робота з текстовими файлами.

Також є клас `BufferedRandom`, успадкований одразу від двох класів: `BufferedReader` та `BufferedWriter`. Методи класу `IOBase` наведено в таблиці 10.2.

Таблиця 10.2 – Методи класу `IOBase`

Категорія	Метод	Опис
Закриття	`close()`	Закриває потік
Буферизація	`flush()`	Очищає буфер потоку (для запису)
Робота з читанням	`read(size=-1)`	Зчитує до `size` байтів або символів (або весь потік, якщо `size` не вказано)
	`readall()`	Зчитує весь потік до кінця
	`readline(size=-1)`	Зчитує один рядок (або максимум `size` байтів/символів)
	`readlines(hint=-1)`	Зчитує всі рядки у список
	`readable()`	Повертає `True`, якщо потік доступний для читання
Робота із записом	`write(data)`	Записує `data` у потік
	`writelines(lines)`	Записує список рядків у потік
	`writable()`	Повертає `True`, якщо потік доступний для запису
Управління позицією в потоці	`seek(offset, whence=SEEK_SET)`	Переміщує позицію в потоці (за зміщенням `offset`)
	`tell()`	Повертає поточну позицію в потоці
	`seekable()`	Повертає `True`, якщо потік підтримує переміщення позиції
Обрізання файлу	`truncate(size=None)`	Обрізає потік до вказаного розміру
Консоль	`isatty()`	Повертає `True`, якщо потік є терміналом (консоллю)

10.5 Модуль os в Python

Модуль `os` дозволяє взаємодіяти з операційною системою на низькому рівні:

– працювати з файлами;

– працювати з директоріями;

– отримувати системну інформацію.

Основні можливості `os` наведено в таблиці 10.3.

Таблиця 10.3 – Можливості модуля `os`

Функція	Опис
`os.name`	Ім'я операційної системи (`'posix'`, `'nt'`)
`os.getcwd()`	Повертає поточний робочий каталог
`os.chdir(path)`	Змінює поточний робочий каталог
`os.listdir(path)`	Повертає список файлів і папок
`os.mkdir(path)`	Створює нову папку
`os.makedirs(path)`	Створює кілька вкладених папок
`os.remove(file)`	Видаляє файл
`os.rmdir(folder)`	Видаляє порожню папку
`os.rename(src, dst)`	Перейменовує або переміщає файл/папку
`os.path`	Підмодуль для роботи зі шляхами

Розглянемо приклади використання цього модуля

Знаходження поточного каталогу:

```
import os

print(os.getcwd())  # Показати де ми зараз
```

Створення папки:

```
os.mkdir('new_folder')
```

Знаходження списку файлів у папці:

```
files = os.listdir('.')
print(files)
```

Перевірка чи існує файл або папка:

```
import os
print(os.path.exists('new_folder'))
```

З'єднання шляхів:

```
full_path = os.path.join('folder', 'file.txt')
print(full_path) # Виведе 'folder/file.txt'
                 # або 'folder\file.txt' залежно від ОС
```

Для *створення* каталогу використовується функція `mkdir()`, в яку передається шлях до цієї папки:

```
import os
# Шлях відносно поточного файлу
os.mkdir("hello")
# Абсолютний шлях
os.mkdir("d:\somedir")
os.mkdir(r"d:\somedir\hello")
```

Але повторний запуск `mkdir()` з тим самим ім'ям викличе `FileExistsError`. Тому замість цього напишемо:

```
if not os.path.isdir(r"d:\somedir"):
    os.mkdir(r"d:\somedir")
```

Для *видалення* каталогу використовується функція `rmdir()`, в яку передається шлях до каталогу:

```
import os
# шлях відносно поточного файлу
os.rmdir("hello")
# абсолютний шлях
os.rmdir(r"d:\somedir\hello")
```

Знайти файли за маскою:

```
import os
# Знайти всі *.py файли
pyfiles = [name for name in os.listdir('somedir')
           if name.endswith('.py')]
```

Робота з шляхами через модуль `os`:

```
import os

# З'єднати частини шляху
path = os.path.join('data', 'file.txt')
```

```
# Перевірити існування файлу
if os.path.exists(path):
    print('Файл існує!')
```

Робота з шляхами через модуль `pathlib`:

```
from pathlib import Path

file_path = Path('data') / 'file.txt'

if file_path.exists():
    print('Файл існує!')
```

10.6 Обробка текстових файлів

Приклад обробки тексту та запису його в файл:

```
# Створення і запис
name=input("Як тебе звати?")
with open('myfile.txt', 'w') as file:
    file.write('Python працює з файлами:\n')
    file.write('-Привіт, '+ name+'\n')
    file.write('-Як твої справи сьогодні?\n')
    hay=input("Як твої справи сьогодні?")
    file.write('-'+hay+'\n')

# Читання
with open('myfile.txt', 'r') as file:
    content = file.read()
    print(content)
```

Результат роботи цієї програми наведено на рисунках 10.2-10.3.

```
Як тебе звати?Світлана
Як твої справи сь◆годні?добре
Python працює з файл◆ми:
-Привіт, Світлана
-Як твої справи сьогодні?
-доб◆е
```

Рисунок 10.2 – Результат програми. Вивід на консоль

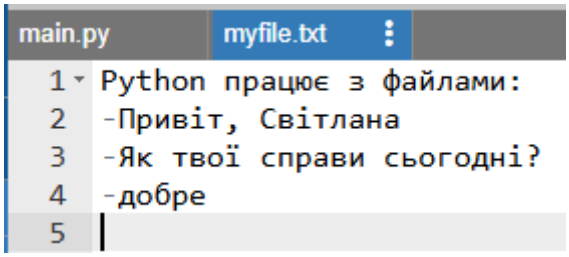


Рисунок 10.3 – Результат програми. Вміст створеного файлу

Ще один приклад роботи з файлами та каталогами в ОС Windows наведено на рисунку 10.4, результат роботи цієї програми на рисунках 10.5-10.6.

```
File  Edit  Format  Run  Options  Window  Help
import os

#Перевіряємо, чи існує папка в якій ми створюватимемо і працюватимемо з файлом
if not os.path.isdir(r"d:\somedir"):
    os.mkdir("d:\somedir")

#Щоб кожного разу не писати шлях до файлу - закинемой його в змінну
dest_file = 'D:\somedir\\first_file.txt'
#Лічильником визначемо скільки разів в файл додаватиметься інформація
counter = int(input("Введіть кількість разів, скільки в файл буде дозаписуватись
i = 1
while i  <= counter:
    textline = f"Ця лінія тексту записана при {i} відкритті файлу"
    with open(dest_file, 'a', newline='\n') as myfile:
        print(textline, file=myfile)
        i= i+1

#Відкриємо файл і переглянемо, що ми записали
with open(dest_file, 'r') as myfile:
    for line in myfile:
        print(line)
```

Рисунок 10.4 – Приклад роботи з файлами та каталогами

```
>>>
==== RESTART: C:/Users/User/AppData/Local/Programs/Python/Python311/12_1.py ====
Введіть кількість разів, скільки в файл буде дозаписуватись якась інформація: 5
Ця лінія тексту записана при 1 відкритті файлу

Ця лінія тексту записана при 2 відкритті файлу

Ця лінія тексту записана при 3 відкритті файлу

Ця лінія тексту записана при 4 відкритті файлу

Ця лінія тексту записана при 5 відкритті файлу
```

Рисунок 10.5 – Результат програми. Вивід на консоль

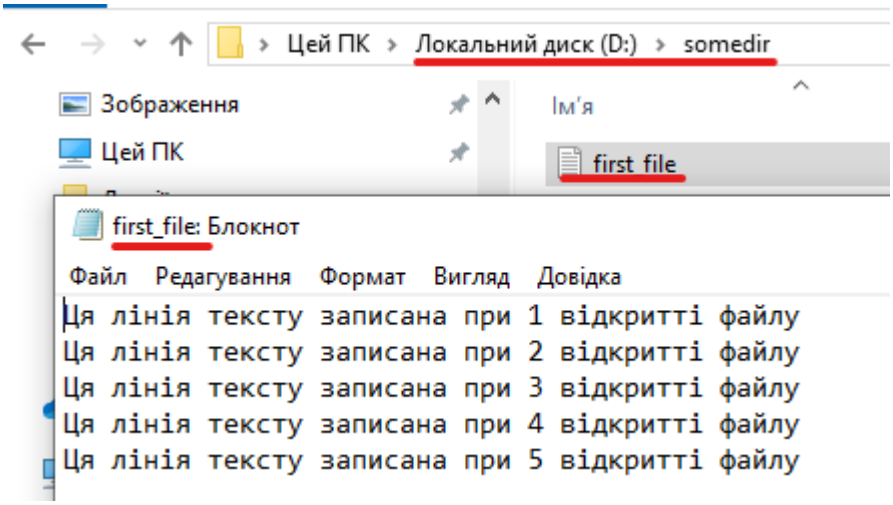


Рисунок 10.6 – Результат програми. Вміст створеного каталогу та файлу

Контрольні питання

1. Що таке файл у контексті програмування на Python?
2. Які типи файлів розрізняють у Python?
3. Як у Python відбувається доступ до файлів із коду програми?
4. Яке призначення функції `open()`? Які параметри вона приймає?
5. Які основні режими відкриття файлів?
6. Що таке файловий потік і яку роль він відіграє у взаємодії з файлами?
7. Як прочитати вміст файлу повністю або построчково?
8. Як виконати запис даних у файл у Python?
9. Чим відрізняється режим `'w'` від `'a'` під час запису у файл?
10. Як правильно закрити файл після роботи з ним і чому це важливо?
11. Яке призначення конструкції `with open(...) as f:` і які її переваги?
12. Як можна перевірити існування файлу перед відкриттям?
13. Як відбувається робота з файлами у різних каталогах?
14. Наведіть приклади практичного використання файлових операцій у Python (зчитування налаштувань, логування, збереження результатів обчислень).

ТЕМА 11. Складність алгоритмів. Методи розробки алгоритмів

11.1 Поняття складності алгоритму. Способи оцінки складності алгоритмів

Оцінка складності алгоритму — це визначення витрат ресурсів (часу виконання та пам’яті) залежно від розміру вхідних даних. Основними способами оцінки складності алгоритмів є:

11.1.1 Асимптотична оцінка складності

Оцінка часу або пам'яті алгоритму в термінах великого O (Big-O нотація).

Основні позначення:

– *O (велике O)* – верхня межа складності (гірший випадок);

– *Ω (велике Омега)* – нижня межа складності (найкращий випадок);

– *Θ (велике Тета)* – точна оцінка складності (середній випадок).

Приклади складностей наведено в таблиці 11.1.

Таблиця 11.1 – Приклади складностей в нотації Big-O

Позначення	Опис	Приклад
O(1)	константна	доступ до елемента масиву
O(log n)	логарифмічна	бінарний пошук
O(n)	лінійна	лінійний пошук
O(n log n)	лінійно-логарифмічна	швидке сортування (QuickSort)
O(n^2)	квадратична	сортування вибором
O(2^n)	експоненційна	перебір підмножин
O(n!)	факторіальна	обхід усіх перестановок

11.1.2 Теоретична оцінка складності

Оцінка кількості основних операцій (порівнянь, присвоєнь, викликів функцій) без реального виконання алгоритму.

Приклад аналізу:

```
for (int i = 0; i < n; i++) {  // Виконується n разів → O(n)
    for (int j = 0; j < n; j++) {  // Виконується n разів → O(n²)
        cout << i << j; // O(1)
    }
}
```

Загальна складність: **O(n^2)**.

11.1.3 Емпірична оцінка складності (експериментальна оцінка)

Вимірювання часу виконання алгоритму (лістинг 11.1) на практиці для різних розмірів вхідних даних.

Методика:

– запуск алгоритму на різних наборах даних;

– вимірювання часу виконання (`std::chrono` в C++);

– побудова графіка залежності часу від розміру `n`.

Лістинг 11.1 – Приклад вимірювання часу в C++

```
#include <iostream>
#include <chrono>

void exampleAlgorithm(int n) {
    for (int i = 0; i < n; i++); // Лінійна складність O(n)
}

int main() {
    int n = 1000000;
    auto start = std::chrono::high_resolution_clock::now();
    exampleAlgorithm(n);
    auto stop = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> duration = stop - start;
    std::cout << "Час виконання: " << duration.count() << "
секунд\n";
    return 0;
}
```

Кінець лістингу 11.1

11.1.4 Оцінка за найгіршим, середнім і найкращим випадком

Найгірший випадок – максимальний час роботи (`O(n²)` для сортування вибором):

– максимальна кількість операцій, яку виконує алгоритм;

– наприклад, у лінійному пошуку `O(n)`, якщо шуканий елемент – останній.

Середній випадок – середній час (`O(n log n)` для швидкого сортування):

– середня кількість операцій для різних входів.

Найкращий випадок – мінімальний час (O(n) для сортування вставками при вже відсортованому масиві):

– мінімальна кількість операцій;

– наприклад, у лінійному пошуку O(1), якщо шуканий елемент перший.

11.1.5 Оцінка використання пам’яті (просторова складність)

Оцінює, скільки додаткової пам’яті потрібно алгоритму. Приклади:

– O(1) – алгоритм використовує фіксовану кількість пам'яті (наприклад, пошук мінімального елемента в масиві);

– O(n) – використання додаткового масиву розміром n (наприклад, збереження результату);

– $O(n^2)$ – використання двовимірної матриці (наприклад, алгоритм Флойда для пошуку найкоротших шляхів).

Отже, оцінка складності алгоритму – ключовий етап його аналізу. Вибір методу залежить від завдання:

– асимптотична оцінка (О-нотація) – для загального аналізу ефективності;

– теоретична оцінка – для підрахунку операцій;

– емпірична оцінка – для тестування реальної продуктивності;

– оцінка за найгіршим/середнім/найкращим випадком – для розуміння поведінки алгоритму;

– просторова складність – для визначення витрат пам’яті.

Мета – вибрати найбільш ефективний алгоритм для конкретної задачі.

11.2 Види складності

Основні види складності:

– *часова складність* – кількість операцій, які виконує алгоритм;

– *просторова складність* – обсяг пам’яті, необхідний для виконання алгоритму;

– *інтелектуальна складність* – зрозумілість алгоритмів і складність їх розробки.

11.2.1 Часова складність

Часто говорять про часову складність алгоритму (швидкодію) – час виконання програми, що працює по даному алгоритму. Часом роботи алгоритму називається кількість виконаних ним елементарних операцій T. Такий підхід дозволяє оцінювати саме якість алгоритму, а не властивості виконавця (наприклад, швидкодія комп'ютера, на якому виконується алгоритм).

Часова складність алгоритму – це час T, необхідний для його виконання в залежності від вихідних даних. Він дорівнює добутку числа елементарних дій до на середній час виконання однієї дії t: T = kt.

Оскільки t залежить від виконавця, який реалізує алгоритм, то природно вважати, що складність алгоритму в першу чергу визначається значенням k. Очевидно, що в найбільшій мірі кількість операцій при виконанні алгоритму залежить від кількості оброблюваних даних. Дійсно, для упорядкування за алфавітом списку з 100 прізвищ потрібно істотно менше операцій, ніж для упорядкування списку з 100 000 прізвищ. Тому складність алгоритму виражають у вигляді функції від обсягу вхідних даних.

Часова складність визначається як кількість елементарних операцій, які виконує алгоритм у залежності від розміру вхідних даних (n). Приклади:

– *O(1) – Константна складність*

```
int getFirstElement(int arr[]) {
    return arr[0];  // Виконується одна операція → O(1)}
```

Алгоритм працює за фіксований час, незалежно від розміру масиву.

– *O(n) – Лінійна складність*

```
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
}
```

Час виконання зростає пропорційно n.

– *O(n²) – Квадратична складність*

```
void printPairs(int arr[], int n) {
```

```
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            std::cout << arr[i] << ", " << arr[j] << "\n";
        }
    }
}
```

Два вкладені цикли – час виконання пропорційний n^2.

11.2.2 Просторова складність

Програмісти зазвичай зосереджують увагу на швидкості алгоритму, але не менш важливі й інші показники – вимоги до обсягу пам’яті, вільного місця на диску. Використання швидкого алгоритму не приведе до очікуваних результатів, якщо для його роботи знадобиться більше пам’яті, ніж є у комп’ютера. Просторова ефективність вимірюється кількістю пам’яті, необхідної для виконання програми.

Комп’ютери мають обмежений обсяг пам’яті. Якщо дві програми реалізують ідентичні функції, то та, яка використовує менший обсяг пам’яті, характеризується більшою просторової ефективністю. Іноді пам’ять стає домінуючим фактором в оцінці ефективності програм. Однак в останні роки в зв’язку з швидким її здешевленням ця складова ефективності поступово втрачає своє значення.

Просторова складність — це кількість пам'яті, яку займає алгоритм під час виконання. Приклади:

– *O(1) – Константна пам’ять*

```
int sum(int a, int b) {
    return a + b; }
```

Використовується лише кілька змінних.

– *O(n) – Лінійне використання пам’яті*

```
int* createArray(int n) {
    return new int[n];  // Виділяється пам’ять на n елементів
}
```

Виділяється пам’ять для n елементів.

Складність алгоритму допомагає оцінити його ефективність. Чим менша складність, тим швидше та менше ресурсів він використовує.

11.2.3 Інтелектуальна складність

При аналізі інтелектуальної складності алгоритму досліджується зрозумілість алгоритмів і складність їх розробки.

Алгоритм швидкого сортування характеризується більшою інтелектуальною складністю в порівнянні з алгоритмом сортування вставками. Якщо запропонувати сотні людей впорядкувати послідовність об'єктів, то найімовірніше, більшість з них використовують алгоритм сортування вибірками. Малоймовірно також, що хтось із них скористається швидким сортуванням. Причини більшої інтелектуальної та просторової складності швидкого сортування очевидні: алгоритм рекурсивний, його досить важко описати, алгоритм довший (мається на увазі текст програми), ніж більш прості алгоритми сортування.

Всі три форми складності зазвичай взаємопов'язані. Як правило, при розробці алгоритму з хорошою часовою оцінкою складності доводиться жертвувати його просторовою і/або інтелектуальною складністю. Завдання можна вирішити швидко, використовуючи великий обсяг пам'яті, або повільніше, займаючи менший обсяг. Типовим прикладом в даному випадку служить алгоритм пошуку найкоротшого шляху. Представивши кару міста у вигляді мережі, можна написати алгоритм для визначення найкоротшої відстані між двома будь-якими точками цієї мережі. Щоб не обчислювати ці відстані щоразу, коли вони нам потрібні, ми можемо вивести найкоротші відстані між усіма точками і зберегти результати в таблиці. Коли нам знадобиться дізнатися найкоротшу відстань між двома заданими точками, ми можемо просто взяти готову відстань з таблиці. Результат буде отриманий миттєво, але це зажадає величезного обсягу пам'яті. Карта великого міста може містити десятки тисяч точок. Тоді, описана вище таблиця, повинна містити більше 10 млрд. комірок. Тобто для того, щоб підвищити швидкодію алгоритму, необхідно використовувати додаткові 10 Гб пам'яті. З цієї залежності виникає ідея об'ємно-

часової складності. При такому підході алгоритм оцінюється, як з точки зорі швидкості виконання, так і з точки зору спожитої пам'яті.

11.3 Порядок складності

Оскільки пам'ять постійно дешевшає, а швидкодія комп'ютерів зростає повільно, головним чином розглядають часову складність T – час виконання програми, що працює по даному алгоритму.

11.3.1 Розрахунок часової складності

Складність алгоритму зазвичай пов'язують з розміром вхідних даних n і визначають як функцію `T(n)`. Наприклад, для алгоритмів обробки масивів в якості розміру n використовують довжину масиву. Функція `T(n)` називається часовою складністю алгоритму. Припустимо, що потрібно вибрати між декількома алгоритмами, які мають різну складність. Який з них краще (працює швидше)? Виявляється, для цього необхідно знати розмір масиву даних, які потрібно обробляти. Порівняємо, наприклад, три алгоритми, складність яких $T1(n) = 10\ 000 \cdot n$, $T2(n) = 100 \cdot n^2$ і $T3(n) = n^3$. Побудуємо ці залежності на графіку (рисунок 11.1). При $n \le 100$ отримуємо $T3(n) < T2(n) < T1(n)$, при $n = 100$ кількість операцій для всіх трьох алгоритмів збігається, а при великих n маємо $T3(n) > T2(n) > T1(n)$.

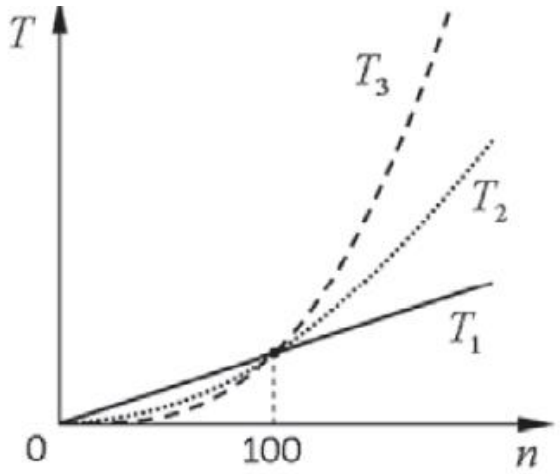


Рисунок 11.1 – Графік залежності часу від розмірності

Розглянемо алгоритми виконання різних операцій з масивом A довжини n, який може бути оголошений в програмі як:

```
int A[n];
```

Приклад 1. Обчислити суму перших трьох елементів масиву (при n ≥ 3).

Вирішення цього завдання містить всього один оператор:

```
S =A[0] + A[1] + A[2];
```

Цей алгоритм включає дві операції додавання і одну операцію запису значення в пам'ять, тому його часова складність T(n) = 3 не залежить від розміру масиву взагалі.

Приклад 2. Обчислити суму всіх елементів масиву.

У цьому завданні вже не обійтися без циклу:

```
S = A[0];
for(i=1;i<n;i++)
S=S + A[i];
```

Тут виконується n - 1 операцій додавання і n операцій запису в пам'ять, тому його складність T(n) = 2n - 1 зростає лінійно зі збільшенням довжини масиву.

11.3.2 Асимптотична складність (порядок складності)

При порівнянні алгоритмів використовується їх асимптотична складність, тобто швидкість зростання кількості операцій при великих значеннях n.

Алгоритм має асимптотичну складність O(f(n)), якщо знайдеться така константа c, що, починаючи з деякого n, виконується умова T(n) ≤ c·f(n).

Це означає, що графік функції c·f(n) йде вище, ніж графік функції T(n), принаймі при n ≥ n0 (рисунок 11.2).

Алгоритм має складність O(f(n)) (порядок складності f(n)), якщо при збільшенні розмірності вхідних даних n, час виконання алгоритму зростає з тією ж швидкістю, що і функція f(n).

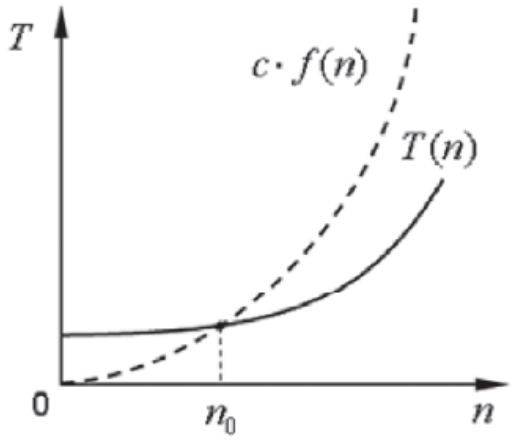


Рисунок 11.2 – Асимптотична складність

11.2.3 Аналіз порядку складності

1. `O(C)` – алгоритм константної складності. Більшість операцій в програмі виконуються тільки раз або тільки кілька разів. Будь-який алгоритм, що завжди вимагає незалежно від розміру даних одного і того ж часу, має константну складність.

Приклад – визначення значення третього елемента масиву, для чого не потрібно ні запам'ятовувати елементи, ні проходити по ним декілька разів. Завжди потрібно просто дочекатися в потоці вхідних даних третій елемент і це буде результатом, на обчислення якого для будь-якої кількості даних потрібний один і той же час.

2. `O(n)` – алгоритм лінійної складності, тобто при збільшенні розміру даних в 10 разів обсяг обчислень збільшується теж приблизно в 10 разів. Такі алгоритми виконуються для операції читання даних і занесення їх в пам'ять. Час роботи програми лінійний, коли кожен елемент вхідних даних потрібно обробити лише лінійну кількість раз.

3. `O(log(n))` і `n * log(n)` – алгоритм логарифмічної складності. Коли час роботи програми логарифмічний, програма починає працювати набагато повільніше зі збільшенням n. Такий час роботи зустрічається зазвичай у програмах, які ділять велику проблему на маленькі і вирішують їх окремо.

4. $O(n^2)$ – алгоритм квадратичної складності, тобто при збільшенні розміру даних в 10 разів, кількість операцій (і час виконання) збільшується приблизно в

100 разів. Алгоритм сортування методом прямого вибору має квадратичну складність $O(n^2)$, так як при сортуванні будь-якого масиву цей алгоритм буде виконувати $(n^2 - n) / 2$ операцій порівняння (при цьому операцій перестановок взагалі може не бути, наприклад, при упорядкованому масиві). Алгоритм сортування бульбашкою виконує два вкладені цикли перебору масиву.

5. $O(n^3)$ – алгоритм кубічної складності. При великих значеннях n алгоритм з кубічного складністю вимагає більшої кількості обчислень, ніж алгоритм зі складністю $O(n^2)$, а той, у свою чергу, працює довше, ніж алгоритм з лінійною складністю. Складність алгоритму множення матриць (таблиць) розміру nxn є $O(n^3)$, так як для обчислення кожного елемента результуючої матриці потрібно n множень і $n-1$ додавань, а всього цих елементів n^2. Припустимо, певним алгоритмом потрібно виконати $4n^3+7n$ умовних операцій, щоб обробити n елементів вхідних даних. При збільшенні n на час роботи буде значно більше впливати піднесення n до кубу, ніж множення його на 4 або ж додавання $7n$.

6. $O(2^n)$ – алгоритм експоненційної (поліноміальної) складності. Такі алгоритми найчастіше виникають в результаті підходу, що називається методом грубої сили.

7. $O(n!)$ – алгоритм факторіальної складності. Зустрічаються найчастіше в задачах оптимізації, які вирішуються тільки методом повного перебору. Найвідоміше завдання такого типу – це завдання комівояжера (бродячого торговця), який повинен відвідати по одному разу кожен із зазначених міст і повернутися в початкову точку. Для нього потрібно вибрати оптимальний маршрут, при якому вартість поїздки (або загальна довжина шляху) буде мінімальною.

Алгоритми зі складністю nC при невеликих значеннях C, наприклад n^2, застосовуються, коли обсяги даних обмежені. Обчислювальна складність алгоритмів, порядок яких визначається функціями C^n і $n!$ дуже велика, тому ці алгоритми придатні тільки для вирішення завдань з дуже малим обсягом інформації, що переробляється.

Основні види алгоритмічної складності, а також час їх виконання в залежності від кількості вхідних параметрів на рисунку 11.3.

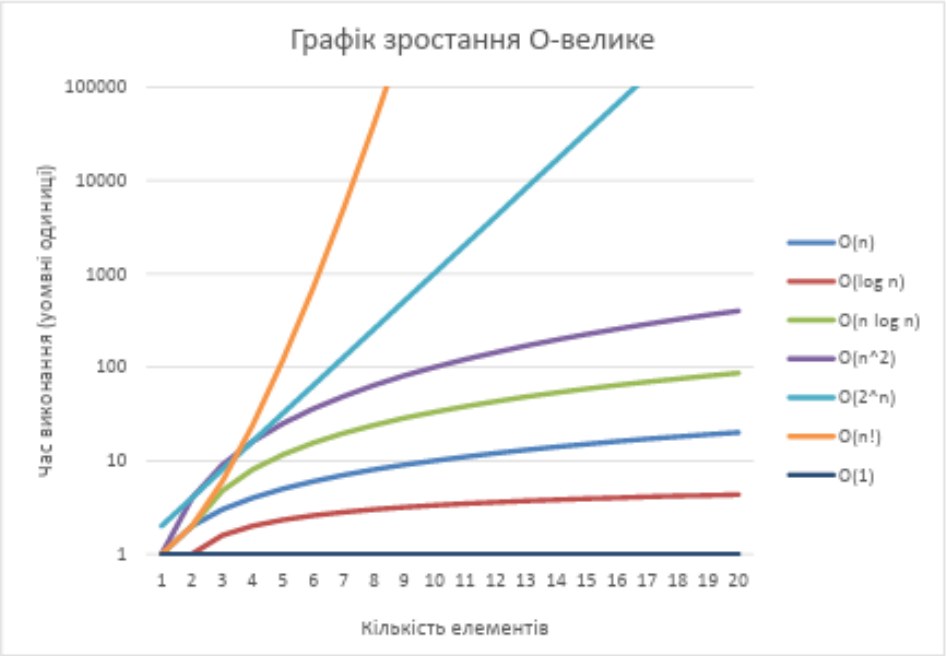


Рисунок 11.3 – Графік росту О-велике

Приклад. Нехай дана послідовність з нулів та одиниць і нам потрібно з'ясувати, чи є там хоч одна одиниця. Яку складність матиме алгоритм розв'язання цієї задачі?

Розв'язання. Нехай n – кількість символів в послідовності. Алгоритм буде послідовно перевіряти, чи немає одиниці в поточному місці заданої послідовності, а потім рухатися далі, поки вхід не скінчиться. Оскільки одиниця дійсно може бути тільки одна, для отримання точної відповіді на це питання в гіршому випадку доведеться перевірити всі n символів входу. Таким чином, алгоритм має складність `O(n)`, іншими словами, він лінійний.

Приклад. Розглянемо код, який для масиву `A[n, n]` знаходить максимальний елемент у кожному рядку.

```
for (i=0;i<n;i++)
{
     max=A[i][0];
     for (j=0;j<n;j++)
          if (A[i][j]>max) max=A[i][j];
     cout<<max;
}
```

У цьому алгоритмі змінна `i` змінюється від `1` до `n`. При кожній зміні `i`, змінна `j` теж змінюється від `1` до `n`. Під час кожної з `n` ітерацій зовнішнього циклу, внутрішній цикл теж виконується `n` раз. Загальна кількість ітерацій внутрішнього циклу дорівнює `n*n`. Це визначає складність алгоритму $O(n^2)$.

Оцінка складності алгоритму допомагає зрозуміти його ефективність.

11.4 Складність популярних алгоритмів

В таблицях 11.2-11.5 наведені часові складності найпопулярніших алгоритмів у форматі Big-O.

Таблиця 11.2 – Алгоритми пошуку

Алгоритм	**Структура даних**	**Часова складність**	**Просторова складність**
Пошук вглибину (DFS)	Граф з $\|V\|$ вершинами і $\|E\|$ ребрами	$O(\|V\| + \|E\|)$	$O(\|V\|)$
Пошук вширину (BFS)	Граф з $\|V\|$ вершинами і $\|E\|$ ребрами	$O(\|V\| + \|E\|)$	$O(\|V\|)$
Бінарний пошук	Відсортований масив з n елементів	$O(\log(n))$	$O(1)$
Лінійний пошук	Масив	$O(n)$	$O(1)$
Пошук у хеш-таблиці	Хеш-таблиця	$O(1)$	$O(1)$
Найкоротший шлях (за алгоритмом Дейкстри) використовуючи двійкову купу як чергу з пріоритетом	Граф з $\|V\|$ вершинами і $\|E\|$ ребрами	$O((\|V\| + \|E\|)\log\|V\|)$	$O(\|V\|)$
Найкоротший шлях (за алгоритмом Дейкстри) використовуючи масив як чергу з пріоритетом	Граф з $\|V\|$ вершинами і $\|E\|$ ребрами	$O(\|V\|^2)$	$O(\|V\|)$

Прмітки: *бінарний пошук працює лише у відсортованих масивах; хеш-таблиці дають O(1) у середньому випадку, але O(n), якщо всі елементи потрапляють в один хеш-осередок (погане хешування).*

Таблиця 11.3 – Алгоритми сортування

Алгоритм	**Структура даних**	**Часова складність**
Швидке сортування	Масив	$O(n^2)$
Сортування злиттям	Масив	$O(n \cdot \log(n))$
Пірамідальне сортування	Масив	$O(n \cdot \log(n))$
Сортування бульбашкою	Масив	$O(n^2)$
Сортування вставками	Масив	$O(n^2)$
Сортування вибором	Масив	$O(n^2)$
Блокове сортування	Масив	$O(n^2)$
Порозрядне сортування	Масив	$O(nk)$

Таблиця 11.4 – Структури даних

Структура даних	**Часова складність**				**Просторова складність**
	Індексація	**Пошук**	**Вставка**	**Видалення**	
Масив	$O(1)$	$O(n)$	-	-	$O(n)$
Динамічний масив	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Однозв’язний список	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Двозв’язний список	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Список з пропусками	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \cdot \log(n))$
Хеш-таблиця	-	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Бінарне дерево пошуку	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Декартове дерево	-	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Б-дерево	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Червоно-чорне дерево	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
АВЛ-дерево	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$

Таблиця 11.5 – Представлення графів

Спосіб представлення	Пам’ять	Додавання вершини	Додавання ребра	Видалення вершини	Видалення ребра	Перевірка суміжності вершин
Список суміжності	$O(\|V\|+\|E\|)$	$O(1)$	$O(1)$	$O(\|V\|+\|E\|)$	$O(\|E\|)$	$O(\|V\|)$
Список інцидентності	$O(\|V\|+\|E\|)$	$O(1)$	$O(1)$	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$
Матриця суміжності	$O(\|V\|^2)$	$O(\|V\|^2)$	$O(1)$	$O(\|V\|^2)$	$O(1)$	$O(1)$
Матриця інцидентності	$O(\|V\| \cdot \|E\|)$	$O(\|V\| \cdot \|E\|)$	$O(\|V\| \cdot \|E\|)$	$O(\|V\| \cdot \|E\|)$	$O(\|V\| \cdot \|E\|)$	$O(\|E\|)$

Примітка. В таблиці 11.5 розглядаємо граф з |V| вершинами та |E| ребрами.

11.5 Формалізація алгоритмів

Формалізація алгоритму – це процес його представлення у строгій, чітко визначеній формі, яка дозволяє однозначно його реалізувати та виконати на комп'ютері.

Основні етапи формалізації алгоритмів:

– *постановка задачі* – визначення початкових даних, очікуваного результату та обмежень;

– *аналіз проблеми* – вибір оптимального методу розв’язку;

– *розробка логіки алгоритму* – опис послідовності дій;

– *представлення алгоритму* – у вигляді блок-схем, псевдокоду або мовою програмування;

– *оцінка коректності та* складності – перевірка правильності алгоритму та аналіз його продуктивності.

Формалізація алгоритму дозволяє зробити його зрозумілим для програмістів, оптимізувати його продуктивність та підготувати до реалізації у програмному коді.

Половина справи зроблена, якщо знати, що поставлена задача має вирішення. В першому наближенні більшість задач, які зустрічаються на практиці, не мають чіткого й однозначного опису. Певні задачі взагалі неможливо сформулювати в термінах, які допускають комп’ютерне вирішення. Навіть якщо допустити, що задача може бути вирішена на комп’ютері, часто для її формального опису потрібна велика кількість різноманітних параметрів. І лише в ході додаткових експериментів можна знайти інтервали зміни цих параметрів.

Якщо певні аспекти вирішуваної задачі можна виразити в термінах якої-небудь формальної моделі, то це, безумовно, необхідно зробити, так як в цьому випадку в рамках цієї моделі можна взнати, чи існують методи й алгоритми вирішення задачі. Навіть якщо такі методи й алгоритми не існують на сьогоднішній день, то застосування засобів і властивостей формальної моделі допоможе в побудові вирішення вихідної задачі.

Практично будь-яку галузь математики або інших наук можна застосувати до побудови моделі певного класу задач. Для задач, числових за своєю природою, можна побудувати моделі на основі загальних математичних конструкцій, таких як системи лінійних рівнянь, диференціальні рівняння. Для задач з символьними або текстовими даними можна застосувати моделі символьних послідовностей або формальних граматик. Вирішення таких задач містить етапи компіляції і інформаційного пошуку.

Коли побудована чи підібрана потрібна модель вихідної задачі, то природно шукати її вирішення в термінах цієї моделі. На цьому етапі основна мета полягає в побудові розв'язку в формі алгоритму, який складається з скінченої послідовності інструкцій, кожна з яких має чіткий зміст і може бути виконана з скінченими обчислювальними затратами за скінчений час. Інструкції можуть виконуватися в алгоритмі будь-яку кількість раз, при цьому вони самі визначають цю кількість повторень. Проте вимагається, щоб при будь-яких вхідних даних алгоритм завершився після виконання скінченої кількості інструкцій. Таким чином, програма, яка написана на основі розробленого алгоритму, при будь-яких початкових даних ніколи не повинна приводити до нескінченних циклічних обчислень.

Є ще один аспект у визначення алгоритмів. Алгоритмічні інструкції повинні мати «чіткий зміст» і виконуватися з «скінченими обчислювальними затратами». Природно, те, що зрозуміло одній людині і має для неї «чіткий зміст», може зовсім інакше представлятися іншій. Те ж саме можна сказати про поняття «скінчених затрат»: на практиці часто важко довести, що при будь-яких вихідних даних виконання послідовності інструкцій завершиться, навіть якщо чітко розуміти зміст кожної інструкції. У цій ситуації, враховуючи всі аргументи за і проти, було б корисним спробувати досягнути узгодження про «скінченні затрати» у відношенні до послідовності інструкцій, які складають алгоритм.

11.6 Покрокове проектування алгоритмів

Покрокове проектування алгоритмів (або *метод поетапної деталізації*) – це метод розробки алгоритмів, при якому складна задача розбивається на простіші підзадачі, що вирішуються окремо.

Оскільки для вирішення вихідної задачі застосовують деяку математичну модель, то тим самим можна формалізувати алгоритм вирішення в термінах цієї моделі. У початкових версіях алгоритму часто застосовуються узагальнені оператори, які потім перевизначаться у вигляді більш дрібних, чітко визначених інструкцій. Але для перетворення неформальних алгоритмів у комп'ютерні програми необхідно пройти через декілька етапів формалізації (цей процес називають покроковою деталізацією), поки не отримають програму, яка повністю складається з формальних операторів мови програмування.

В основу процесу проектування програми з розбиттям її на достатню кількість дрібних кроків можна покласти наступну послідовність дій:

1. Вихідним станом процесу проектування є більш-менш точне формулювання мети алгоритму, або конкретніше – результату, який повинен бути отриманий при його виконанні. Формулювання проводиться на природній мові.

2. Проводиться збір фактів, які стосуються будь-яких характеристик алгоритму, і робиться спроба їх представлення засобами мови.

3. Створюється образна модель процесу, який відбувається, використовуються графічні та інші способи представлення, образні «картинки», які дозволяють краще зрозуміти виконання алгоритму в динаміці.

4. В образній моделі виділяється найбільш суттєва частина, для якої підбирається найбільш точне словесне формулювання.

5. Проводиться визначення змінних, які необхідні для формального представлення даного кроку алгоритму.

6. Вибирається одна з конструкцій – проста послідовність дій, умовна конструкція або цикл. Складні частини вибраної формальної конструкції (умова, заголовок циклу) залишаються в словесному формулюванні.

7. Для інших неформалізованих частин алгоритму, які залишились у словесному формулюванні – перерахована послідовність дій повторюється.

Зовнішня сторона такого процесу проектування програми полягає в тому, що одне словесне формулювання алгоритму замінюється на одну з трьох можливих конструкцій мови, елементи якої продовжують залишатися в неформальному, словесному формулюванні. Проте, це зовнішня сторона процесу. Насправді, кожне словесне формулювання алгоритму містить важливий перехід, який виконується в голові програміста і не може бути формалізованим – перехід від мети (результату) роботи програми до послідовності дій, які приводять до потрібного результату. Тобто алгоритм вже формулюється, але тільки з використанням образного мислення і природної мови.

На першому етапі створюється модель вихідної задачі, для чого застосовуються відповідні математичні моделі. На цьому етапі для знаходження рішення також будується неформальний алгоритм.

На наступному етапі алгоритм записується на псевдомові – композиції конструкцій мови програмування і менш формальних і узагальнених операторів на простій мові. Продовженням цього етапу є заміна неформальних операторів послідовністю більш детальних і формальних операторів. З цієї точки зору програма на псевдомові повинна бути достатньо детальною, так як в ній фіксуються (визначаються) різні типи даних, над якими виконуються оператори. Потім створюються абстрактні типи даних для кожного зафіксованого типу даних (за винятком елементарних типів даних, таких як цілі й дійсні числа або символьні стрічки) шляхом завдання імен функцій для кожного оператора, який працює з даними абстрактного типа, і заміни їх (операторів) викликом відповідних функцій.

Третій етап процесу – програмування – забезпечує реалізацію кожного абстрактного типа даних і створення функцій для виконання різних операторів над даними цих типів. На цьому етапі також замінюються всі неформальні оператори псевдомови на код мови програмування. Результатом цього етапу повинна бути виконувана програма. Після її наладки отримують працюючу програму.

Розглянемо приклад покрокового проектування задачі знаходження максимального елементу у масиві чисел.

1. Декомпозиція:

1) Прочитати вхідні дані (масив чисел).
2) Ініціалізувати змінну для збереження максимального значення.
3) Пройти по всіх елементах масиву.
4) Якщо поточний елемент більший за збережене значення – оновити його.
5) Вивести результат.

2. Покрокова деталізація у вигляді псевдокоду:

```
1. Вхід: масив A розміру n
2. max = A[0]  // Припустимо, що перший елемент є найбільшим
3. Для i від 1 до n-1:
      Якщо A[i] > max:
          max = A[i]
4. Вивести max
```

Цей метод покрокового проектування допомагає створювати ефективні та добре структуровані алгоритми, які легко реалізовувати у коді.

11.7 Метод частинних цілей

Метод частинних цілей – це підхід до розв’язання складних задач шляхом їхнього розбиття на менші, простіші підзадачі (*частинні цілі*), які можна вирішити незалежно або послідовно, щоб досягти загальної мети.

Основні принципи методу частинних цілей

1) Розбиття задачі – поділ складної проблеми на логічно незалежні підзадачі.

2) Формулювання частинних цілей – визначення конкретних проміжних результатів, які ведуть до загальної мети.

3) Послідовне або паралельне розв’язання підзадач – залежно від їхньої природи.

4) Об’єднання отриманих результатів – узагальнення вирішених підзадач для отримання підсумкового рішення.

Цей метод полягає у тому, що глобальна велика задача ділиться (якщо це можливо) на окремі задачі. Якщо велику задачу, можливо, і не можна осягнути, зрозуміти, як її розв'язувати, то для кожної з поділених задач може існувати давно відомий алгоритм розв'язку, або пошук такого алгоритму є значно легшою задачею.

Наприклад задача сортування масиву прямим обміном. Алгоритм сортування зводиться до відшукання мінімуму у несортованій частині масиву та дописування його до сортованої частини.

Функція пошуку проекції точки на площину зводиться до двох:

– пошуку рівняння прямої, перпендикулярної площині та такої, що проходить через задану точку;

– пошуку точки перетину перпендикуляру та площини.

Щоб знайти довжину перпендикуляру від точки до прямої, тобто відстань від прямої до площини, слід після знаходження проекції точки на площину звернутися до функції пошуку відстані між двома точками.

Останній приклад показує, що існують випадки, коли розбиття задачі на частинні задачі може призвести до зайвого ускладнення алгоритму, збільшення часу його роботи. Відстань від площини до точки шукається за допомогою канонічного рівняння площини, якщо у нього підставити координати точки, «за одну дію». Але метод частинних цілей дає добре структурований, часто більш наочний алгоритм, тому кожного разу треба окремо вирішувати, яким шляхом іти.

Усі рекурсивні алгоритми є реалізацією методу частинних цілей.

Розглянемо приклад застосування для задачі впорядкування списку студентів за зростанням оцінок.

1. Виділення частинних цілей:

1) Отримати вхідні дані (список студентів та їхні оцінки).
2) Вибрати алгоритм сортування.
3) Реалізувати сортування.
4) Вивести відсортований список.

2. Реалізація частинних цілей у псевдокоді:

```
1. Отримати список студентів з оцінками.
2. Вибрати метод сортування (наприклад, швидке сортування).
3. Виконати сортування за оцінками у порядку зростання.
4. Вивести результати.
```

3. Об'єднання частинних цілей:

Після виконання всіх підзадач ми отримуємо відсортований список студентів.

Переваги методу частинних цілей:

– спрощує розв'язання складних задач;

– робить алгоритм більш структурованим;

– дозволяє використовувати вже розроблені рішення для підзадач

– полегшує тестування та налагодження.

Метод частинних цілей широко використовується в програмуванні, алгоритмічному мисленні та розробці програмного забезпечення.

11.8 Динамічне програмування

Динамічне програмування (Dynamic Programming, DP) – це метод оптимізації, що використовується для розв'язання задач, які можна розбити на перекриваючі підзадачі. Замість повторного розв'язання однакових підзадач, їхні результати зберігаються і використовуються повторно.

Нерідко не вдається поділити задачу на невелику кількість задач меншого розміру, об'єднання розв'язків яких дозволить отримати рішення початкової задачі. У таких випадках пробують поділити задачу на стільки задач, скільки необхідно, потім кожну поділену задачу ділять на ще декілька менших і так далі. Якщо б весь алгоритм зводився саме до такої послідовності дій, то в результаті отримали б алгоритм з експоненціальним часом виконання.

Але часто вдається отримати лише поліноміальну кількість задач меншого розміру і тому ту чи іншу задачу доводиться вирішувати багаторазово. Якщо б замість того відслідковувати рішення кожної вирішеної задачі і просто шукати у

випадку необхідності відповідний розв’язок, то отримують алгоритм з поліноміальним часом виконання.

З точки зору реалізації, іноді, буває простіше створити таблицю розв’язків усіх задач меншого розміру, які доведеться вирішувати. Заповнюють таку таблицю незалежно від того, чи потрібна в реальному випадку конкретна задача для отримання загального розв’язку. Заповнення таблиці складових задач для отримання розв’язку певної задачі отримало назву динамічне програмування.

Динамічним програмуванням (в найбільш загальній формі) називають процес покрокового розв’язку задачі, коли на кожному кроці вибирається один розв’язок з множини допустимих на цьому кроці, причому такий, який оптимізує задану цільову функцію або функцію критерію. В основі теорії динамічного програмування лежить принцип оптимальності Белмана.

Форма алгоритму динамічного програмування може бути різною – спільною їх темою є лише заповнення таблиці і порядок заповнення її елементами.

Основні ідеї динамічного програмування:

1) Оптимальний підхід – розв’язок задачі будується на основі розв’язків її частинних підзадач.

2) Запам’ятовування (Memoization) – зберігання проміжних результатів для уникнення повторних обчислень.

3) Розбиття на підзадачі – задача розбивається на менші підзадачі, які можуть повторюватися.

4) Кроковий розв’язок (Bottom-up) – обчислення починається з найпростіших випадків і поступово розширюється до повного рішення.

Методи реалізації динамічного програмування:

1) Мемоізація **(**Top-Down**)** – рекурсивний підхід із запам’ятовуванням вже обчислених результатів у таблиці.

2) Табличний підхід (Bottom-Up) – ітеративне заповнення таблиці розв’язків від найменших підзадач до основної задачі.

Розглянемо приклад задачі з динамічного програмування.

*Задача***.** Знайти N-е число Фібоначчі (де $F(n) = F(n-1) + F(n-2)$).

1) Рекурсивний підхід (неефективний, $O(2^n)$)

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Проблема: багаторазові обчислення однакових значень.

2) Динамічне програмування з мемоізацією (Top-Down, O(n))

```
#include <vector>
using namespace std;

vector<int> memo(100, -1); // Ініціалізація кешу

int fibonacci(int n) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n]; // Перевірка кешу
    return memo[n] = fibonacci(n-1) + fibonacci(n-2);
}
```

Зберігає значення, щоб не обчислювати їх повторно.

3) Табличний підхід (Bottom-Up, O(n))

```
int fibonacci(int n) {
    vector<int> dp(n + 1);
    dp[0] = 0, dp[1] = 1;
    for (int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}
```

Цей підхід виключає рекурсію та оптимізує використання пам’яті.

Приклади задач, що розв’язуються за допомогою динамічного програмування:

– числа Фібоначчі;

– задача про рюкзак (*Knapsack problem*);

– найдовша спільна підпослідовність (*Longest Common Subsequence, LCS*);

– найкоротший шлях у зваженому графі (*Floyd-Warshall*);

– задача про розбиття числа на мінімальну кількість квадратів (*Perfect Squares*).

Переваги динамічного програмування:

– значно скорочує кількість обчислень у задачах з повторюваними підзадачами;

– використовує збереження проміжних результатів (ефективне використання ресурсів);

– дозволяє оптимально розв’язувати складні комбінаторні задачі.

Динамічне програмування є потужним інструментом, який використовується в алгоритмічних змаганнях, обробці даних та оптимізаційних задачах.

11.9 Метод сходження

Даний метод полягає у тому, щоби протягом пошуку найкращого розв’язку алгоритм відшукував все кращі та кращі варіанти розв’язку. Якщо ввести деяку кількісну оцінку якості розв’язку, який шукається, то такий метод подібний на здолання все нової та нової висоти при сходженні на вершину.

Розглянемо як приклад задачу про мандрівного крамаря. Крамар має проїхати n міст по циклу, використавши для цього цикл найменшої довжини та побувавши у кожному місті по одному разу.

Розв’язок можна шукати різними шляхами. Але самим очевидним є відшукати спочатку просто який-небудь цикл, що може вважатися «хорошим». Для цього, наприклад, можна з першого міста їхати у найближче, звідти – у найближче з тих, що залишилися і так далі. З останнього міста треба буде повернутися назад. Це – перша вершина. Далі можна спробувати поліпшити цей варіант. Якщо це вийде, то це буде нова висота. Найкоротший цикл шукається за допомогою досить складних алгоритмів.

Ще один приклад використання методу сходження – ітерації.

Знайти значення корінь квадратний для довільного дійсного x – u = sqrt(x).

Якщо `u = sqrt(x)`, то `x/u = u`;

Якщо `u < sqrt(x)`, то `x/u > sqrt(x)`.

Тому будь-коли середнє між u та x/u ближче до sqrt(x), ніж u. Тому, почавши з будь-якого «близького» початкового значення u(0), можна поступово поліпшувати наближення: u(n+1) := (u(n) + x / u(n)) / 2 доти, поки не буде виконуватися умова |u(n)- x / u(n)| < eps.

Даний тип алгоритмів має ще одну назву – «скупі» алгоритми. На кожній окремій стадії «скупий» алгоритм вибирає той варіант, який є локально оптимальним в тому чи іншому змісті. Раніше вже були розглянуті різні «скупі» алгоритми – алгоритм побудови найкоротшого шляху Дейкестри і алгоритм побудови стовбурного дерева мінімальної вартості Крускала. Алгоритм найкоротшого шляху Дейкестри є «скупим» у тому розумінні, що він вибирає вершину, найближчу до джерела, серед тих, найкоротший шлях яких ще невідомий. Алгоритм Крускала також «скупий» – він вибирає із ребер, які залишилися і не створюють цикл, ребро з мінімальною вартістю.

Потрібно зауважити, що не кожний «скупий» алгоритм дозволяє отримати оптимальний результат в цілому. Як це буває у житті, «скупа стратегія» у більшості випадків забезпечує локальний оптимум, у той же час як в цілому результат буде неоптимальним.

Узагальнюючи вище сказане, можна зробити висновок – стратегія методу полягає в тому, щоб почати з випадкового рішення і потім робити послідовні наближення. Почавши з випадково вибраного рішення, алгоритм робить випадковий вибір. Якщо нове рішення краще попереднього, програма закріплює зміни і продовжує перевірку інших випадкових змін. Якщо зміна не покращує рішення, програма відкидає його і робить нову спробу.

11.10 Дерева розв'язків

Багато складних реальних задач можна змоделювати за допомогою дерев розв'язків. Кожний вузол дерева представляє один крок вирішення задачі. Кожна гілка в дереві представляє розв'язок, який веде до більш повного рішення. Листки є остаточним розв'язком. Мета полягає в тому, щоб знайти «найкращий

шлях» від кореня дерева до листка при виконанні певних умов. Ці умови і значення поняття «найкращий» для шляху залежить від конкретної задачі.

Стратегію настільних ігор, таких як шахи, шашки, або «хрестики-нулики» можна змоделювати за допомогою дерев гри. Якщо в який-небудь момент гри існує 30 можливих ходів, то відповідний вузол у дереві гри матиме 30 гілок. Наприклад, для гри в «хрестики-нулики» кореневий вузол відповідає початковій позиції, при якій дошка порожня. Перший гравець може помістити хрестик в будь-яку з дев'яти кліток дошки. Кожному з цих дев'яти можливих ходів відповідає гілка дерева, яка виходить з кореня. Дев'ять вузлів на кінці цих гілок відповідають дев'яти різним позиціям після першого ходу гравця.

Після того, як перший гравець зробив хід, другий може поставити нулик в будь-яку з восьми кліток, що залишилися. Кожному з цих ходів відповідає гілка, що виходить з вузла, який відповідає поточній позиції.

Як можна здогадатися, дерево гри навіть для такої простої гри росте дуже швидко. Якщо воно буде продовжувати рости таким чином, що кожний наступний вузол в дереві матиме на одну гілку менше попереднього, то повне дерево гри матиме 9! = 362880 листки. Тобто, в дереві буде 362880 можливих шляхів розв'язку, які відповідають можливим варіантам гри.

Насправді багато з вузлів дерева в реальній грі будуть відсутніми, оскільки відповідні їм ходи заборонені правилами гри. Якщо гравець, що ходив першим, за три свої ходи поставить хрестики у верхній лівій, верхній середній і верхній правій клітках, то він виграє і гра закінчиться. Вузол, який відповідає цій позиції, не матиме нащадків, оскільки гра завершується на цьому кроці.

Після вилучення всіх неможливих вузлів в дереві залишається біля четверті мільйона листків. Це все ще дуже велике дерево, і пошук у ньому оптимального розв'язку методом повного перебору займає достатньо багато часу. Можна ще скоротити розмір цього дерева, враховуючи симетричність деяких позиція, але це підходить лише для конкретної гри. Для складніших ігор, таких як шашки, шахи або го, дерева мають величезний розмір. Якби під час кожного ходу в шахах гравець мав би 16 можливих варіантів, то дерево гри мало б більше трильйона вузлів після п'яти ходів кожного з гравців.

11.12 Програмування з поверненнями назад

Іноді доводиться мати справу з задачами пошуку оптимального розв’язку, коли неможливо застосувати жоден з відомих алгоритмів, які здатні допомогти відшукати оптимальний варіант розв’язку, і залишається застосувати останній засіб – повний перебір.

Багато задач не припускають аналітичного розв'язку, а тому їх залишається розв'язувати методом спроб та помилок, тобто перебираючи фактично варіанти та відкидаючи їх у разі невдачі. У разі, якщо побудова розв'язку є складною процедурою, то фактично під час роботи будується дерево можливих кроків алгоритму, а потім - у разі невдачі - відрізаються відповідні гілки дерева, доки не буде побудовано того шляху, що веде до успіху. Проходження вздовж гілок дерева та відходження у разі невдач і є алгоритм з поверненнями.

Приклади використання:

– генерація перестановок та комбінацій (наприклад, знаходження всіх можливих паролів);

– розв’язування головоломок (судоку, шахові задачі – ферзі, коні);

– пошук у графах (лабіринти, розфарбовування графів);

– оптимізаційні задачі (розбиття чисел, укладання рюкзака).

11.13 Евристичні алгоритми

У складніших іграх практично неможливо провести пошук навіть в невеликому фрагменту дерева. У цих випадках, можна використовувати різні евристики. Евристикою називається алгоритм або емпіричне правило, яке ймовірно, але не обов’язково дасть добрий результат.

Наприклад, в шахах звичайною евристикою є «посилення переваги». Якщо у супротивника менше сильних фігур і однакова кількість інших, то слід йти на розмін при кожній нагоді. Зменшення кількості фігур робить дерево рішень коротшим і може збільшити відносну перевагу. Ця стратегія не гарантує виграшу, але підвищує його вірогідність.

Інша евристика, що часто використовується, полягає в присвоєнні різних ваг різним клітинкам. У шахах вага кліток в центрі дошки вища, оскільки фігури, що знаходяться на цих позиціях, можуть атакувати більшу частину дошки. Коли обчислюється вага поточної позиції на дошці, вона може присвоюватися більшою фігурам, які займають клітки в центрі дошки.

Якщо якість рішення не так важлива, то прийнятним може бути результат, отриманий за допомогою евристики. В деяких випадках точність вхідних даних може бути недостатньою. Тоді хороше евристичне рішення може бути таким же правильним, як і теоретично «якнайкраще рішення».

У попередньому прикладі метод гілок і границь використовувався для вибору інвестиційних можливостей. Проте, вкладення можуть бути ризикованими, і точні результати часто наперед невідомі. Можливо, що наперед буде невідомий точний дохід або навіть вартість деяких інвестицій. В цьому випадку, ефективне евристичне рішення може бути таким же надійним, як і якнайкраще рішення, яке ви може обчислити точно.

Отже, евристичні алгоритми – це алгоритми, які мають такі властивості:

Вони дозволяють знайти добрі, хоча і не завжди найкращі розв’язки з усіх, що існують.

Метод пошуку або побудови розв’язку звичайно значно простіший, ніж той що гарантує оптимальність розв’язку.

Поняття «добрий розв’язок» змінюється від задачі до задачі, тому його важко визначити точно. Припустимість використання евристики залежить від співвідношення часу та складності пошуку розв’язку обома способами та співвідношення якості обох розв’язків.

Контрольні питання

1. Що таке складність алгоритму?
2. Які є основні способи оцінки складності алгоритму?
3. У чому різниця між часовою та просторовою складністю?
4. Які види складності алгоритмів існують?
5. Що означає позначення O(1)? Наведіть приклад.
6. Як працює алгоритм зі складністю O(log n)?
7. Який порядок складності має лінійний пошук?

8. Що означає асимптотична оцінка складності алгоритму?
9. Поясніть поняття порядку складності (О-нотація).
10. Які основні класи складності алгоритмів і що вони означають?
11. Наведіть приклади алгоритмів із різними порядками складності.
12. Як визначити часову складність циклу або вкладених циклів?
13. Які існують способи формалізації алгоритмів?
14. У чому полягає покрокове проєктування алгоритмів?
15. Опишіть суть методу частинних цілей у процесі розробки алгоритмів.
16. Що таке динамічне програмування і для яких задач воно ефективне?
17. У чому полягає суть методу сходження?
18. Що являє собою дерево розв’язків (дерево пошуку) при проєктуванні алгоритмів?
19. Що таке евристичні алгоритми, у яких випадках вони використовуються, і які приклади можна навести?

СПИСОК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ

1. Python Essentials 1. Cisco Networking Academy: Learn Cybersecurity, Python & More. URL: https://www.netacad.com/courses/python-essentials-1?courseLang=uk-UA (дата звернення: 15.05.2025).

2. Python Essentials 2. Cisco Networking Academy: Learn Cybersecurity, Python & More. URL: https://www.netacad.com/courses/python-essentials-2?courseLang=en-US (дата звернення: 15.05.2025).

3. Python. Python documentation. URL: https://docs.python.org/uk/3.13/tutorial/index.html (дата звернення: 10.06.2025).

4. Бхаргава А. Грокаємо алгоритми. Ілюстрований посібник для програмістів і допитливих. ArtHuss, 2024. 256 с.

5. Васильєв О. М. Програмування в Python. Теорія і практика : навч. посіб. Київ : Видавництво Ліра-К, 2023. 462 с.

6. Маттес Е. Пришвидшений курс Python. Львів: «Видавництво Старого Лева», 2021. 600 с.

7. Програмування для всіх: основи Python. Prometheus. https://apps.prometheus.org.ua/learning/course/course-v1:Michigan+PFE101+2023_T3/home (дата звернення: 16.05.2025).

8. Програмування Українською. #1 Python з нуля. Налаштування середовища. Hello world. YouTube. URL: https://www.youtube.com/watch?v=wHkKFes6izw (дата звернення: 15.05.2025).

9. Школа програмування. Уроки Python з нуля / #1 – Програмування на Пітон для початківців. YouTube. URL: https://www.youtube.com/watch?v=IyHTJTLqkCA (дата звернення: 28.05.2025).

ІНФОРМАЦІЙНІ РЕСУРСИ

1. Visualizing Algorithms. URL: https://bost.ocks.org/mike/algorithms/ (дата звернення: 29.05.2025).

2. Інструмент для створення блок-схем URL: https://app.diagrams.net/ (дата звернення: 25.05.2025).

3. Он-лайн компілятор URL: https://www.onlinegdb.com/ (дата звернення: 25.05.2025).

4. W3Schools.com. W3Schools Online Web Tutorials. URL: https://www.w3schools.com/python/ (дата звернення: 15.05.2025).

А45 **Алгоритмізація та програмування**: конспект лекцій (частина друга) для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Інформаційні системи та технології охорони і безпеки» галузь знань F Інформаційні технології спеціальності F6 Інформаційні системи та технології денної та заочної форм навчання / уклад. С. В. Лавренчук. Луцьк: ЛНТУ, 2025. 248 с.

Конспект лекцій (частина друга) з дисципліни «Алгоритмізація та програмування» складений відповідно до діючої програми курсу.

Призначений для здобувачів вищої освіти спеціальності Інформаційні системи та технології освітньої програми «Інформаційні системи та технології охорони і безпеки»

Комп’ютерний набір С. В. Лавренчук

Редактор С. В. Лавренчук

Підп. до друку «___» _____ 2025р.
Формат 60х84/16. Папір офс. Гарнітура Таймс.
Ум. друк. арк. ____. Тираж 10 прим. Зам. ____

Луцький національний технічний університет
43018, м. Луцьк, вул. Львівська, 75