

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
МІСЬКОГО ГОСПОДАРСТВА імені О. М. БЕКЕТОВА**

Р. В. Воронов, В. А. Герасименко

ОСНОВИ ПРОГРАМУВАННЯ

КОНСПЕКТ ЛЕКЦІЙ

*(для здобувачів першого (бакалаврського) рівня вищої освіти денної та
заочної форм навчання зі спеціальності G3 – Електрична інженерія)*

**Харків
ХНУМГ ім. О. М. Бекетова
2026**

УДК 004.43

Воронов Р. В. Основи програмування : конспект лекцій для здобувачів першого (бакалаврського) рівня вищої освіти денної та заочної форм навчання зі спеціальності G3 – Електрична інженерія) / Р. В. Воронов, В. А. Герасименко ; Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. – Харків : ХНУМГ ім. О. М. Бекетова, 2026. – 105 с.

Автори:

канд. техн. наук, доц. Р. В. Воронов,
канд. техн. наук, доц. В. А. Герасименко

Рецензент

В. Є. Плюгін, доктор технічних наук, професор, завідувач кафедри систем електропостачання та електроспоживання міст Харківського національного університету міського господарства імені О. М. Бекетова

*Рекомендовано кафедрою світлотехніки і джерел світла,
протокол № 1 від 25.08.2025*

© Р. В. Воронов, В. А. Герасименко, 2026

© ХНУМГ ім. О. М. Бекетова, 2026

ЗМІСТ

ВСТУП.....	4
1 БАЗОВІ ПОНЯТТЯ МОВИ <i>PYTHON</i>	10
1.1 Основні визначення.....	10
1.2 Встановлення <i>Python</i>	14
2 АЛГОРИТМІЧНІ СТРУКТУРИ.....	16
2.1 Поняття алгоритму та його фундаментальні властивості.....	16
2.2 Умовні оператори.....	21
2.3 Циклічні оператори	25
3 ФУНКЦІЇ.....	29
3.1 Загальна інформація про функції в <i>Python</i>	29
3.2 Синтаксис оголошення та виклику функцій	30
3.3 Параметри та аргументи функцій.....	32
3.4 Простори імен та області видимості	35
3.5 Оператор <i>return</i>	38
3.6 Анонімні функції: функція <i>lambda</i>	39
4 ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ.....	40
4.1 Вступ до об'єктно-орієнтованого програмування.....	40
4.2 Класи, об'єкти та методи.....	44
4.3 Конструктори, деструктори	48
4.4 Спадкування.....	50
5 СТРУКТУРИ ДАНИХ	56
5.1 Списки	56
5.2 Кортежі.....	63
5.3 Словники	65
5.4 Множини	69
6 МОДУЛІ ТА БІБЛІОТЕКИ	76
6.1 Загальна інформація про модулі та бібліотеки	76
6.2 Модулі стандартної бібліотеки	80
6.3 Сторонні (зовнішні) бібліотеки	86
7 РОБОТА З ФАЙЛАМИ. ВИНЯТКИ.....	96
7.1 Основні операції з файлами	96
7.2 Винятки	100
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ	104

ВСТУП

Оскільки існує велика кількість мов програмування, питання вибору є одним із перших, яке постає перед початківцями. Саме тому виникає потреба зрозуміти, яку мову доцільно обрати для навчання. З огляду на те, що сьогодні мову *Python* використовують мільйони людей у всьому світі, неможливо надати однозначну відповідь на питання щодо вибору мови програмування. Вибір інструментів розробки зазвичай залежить від конкретних умов, визначених завдань і особистих уподобань розробника.

У межах дисципліни «Основи програмування» обрано мову *Python* як базову, оскільки вона поєднує простоту синтаксису, широкі можливості застосування та високу популярність у різних галузях науки й техніки. Це робить її ефективним інструментом для формування базових навичок програмування та подальшого їх практичного використання.

Python – це інтерпретована, об'єктно-орієнтована та високорівнева мова програмування загального призначення з динамічною типізацією та автоматичним управлінням пам'яттю, орієнтована на підвищення продуктивності розробника, читабельності коду, а також на забезпечення імпортування написаних на ній програм.

Python є відносно простим у вивченні та водночас потужним інструментом програмування. Він надає ефективні високорівневі структури даних, а також пропонує зручний і водночас результативний підхід до об'єктно-орієнтованого програмування. Елегантний синтаксис мови, її динамічна типізація та інтерпретований характер роблять *Python* особливо придатним для написання сценаріїв і швидкої розробки програмних застосунків у різних предметних областях і на більшості сучасних платформ. Мова *Python* застосовується для вирішення широкого спектра завдань у різних галузях діяльності завдяки своїй універсальності, простоті використання та великій кількості бібліотек. Вона широко використовується для розробки вебзастосунків, програмного забезпечення, автоматизації

різноманітних завдань, створення внутрішніх інструментів, а також у сфері тестування та системного адміністрування. Окремо варто відзначити її значну роль у візуалізації та аналізі даних, машинному навчанні та наукових дослідженнях, де *Python* є одним із провідних інструментів для обробки інформації, математичного моделювання та проведення експериментів. Така багатофункціональність робить *Python* ефективним засобом як для навчання програмуванню, так і для вирішення практичних інженерних і наукових завдань.

Мову програмування *Python* було створено нідерландським програмістом Гвідо ван Россумом у період його роботи в Нідерландському науково-дослідному інституті. До цього він брав участь у розробці мови *ABC*, яка представлялась як проста та зрозуміла декларативна мова програмування. Попри цікаві концептуальні рішення, мова *ABC* не набула поширення, що стало однією із причин створення нової мови. У грудні 1989 року Гвідо ван Россум розпочав роботу над власним проєктом – мовою *Python*, яка частково успадкувала ідеї *ABC*, але була позбавлена її недоліків. Офіційною датою появи *Python* вважається 20 лютого 1991 року. Протягом тривалого часу Гвідо ван Россум відігравав ключову роль у розвитку мови, тобто його слово було вирішальним у зміні і розвитку *Python*. Назва мови *Python* походить від британського гумористичного телешоу *Monty Python's Flying Circus*, що відображає прагнення автора зробити мову не лише потужною, але й доступною та легкою для сприйняття. Оскільки *Python* було створено пізніше, ніж багато інших мов програмування, під час його розробки було враховано кращі ідеї та підходи, запозичені з різних мов, що значною мірою сприяло його популярності та ефективності.

Мова програмування *Python* має низку суттєвих переваг, які зумовлюють її широку популярність і ефективність для практичного застосування:

1. Простота синтаксису. Пайтон вирізняється простим зрозумілим кодом. За прочитанням він нагадує чіткий текст англійською. Із синтаксису

прибрано все зайве, тому в ньому немає непотрібних виразів та дужок. На *Python* можна написати менше рядків, ніж на *Java* або *C/C++* для виконання одного й того самого завдання: чим простіше, тим краще.

2. Мова легка в освоєнні та проста для вивчення. Це зумовлено зрозумілим і логічним синтаксисом, завдяки цьому початківці можуть швидко опанувати базові принципи програмування та зосередитися на вирішенні практичних завдань, а не на складнощах мовної реалізації.

3. Вільна і відкрита мова. *Python* є прикладом вільного та відкритого програмного забезпечення (*FLOSS – Free / Libre and Open Source Software*). Це означає, що користувачі мають можливість вільно використовувати мову програмування, отримувати доступ до її вихідного коду, вивчати його, модифікувати та застосовувати окремі компоненти у власних програмних проєктах. Завдяки такій відкритій моделі розвитку *Python* постійно вдосконалюється спільнотою розробників усього світу. Це забезпечує його актуальність, стабільність та швидке впровадження нових можливостей, що сприяє подальшому розвитку мови як сучасного інструменту програмування.

4. Наявність єдиного стилістичного стандарту написання коду – *PEP 8*. Він визначає правила оформлення програм, зокрема структуру коду, відступи, іменування змінних, розташування операторів та інші елементи стилю. Завдяки дотриманню *PEP 8* код стає більш читабельним, зрозумілим і уніфікованим незалежно від того, хто його написав. Це суттєво полегшує командну розробку, супровід програмного забезпечення та зменшує кількість помилок, пов'язаних із неоднозначністю стилю написання.

5. Висока продуктивність розробки програмного забезпечення. Цього досягають завдяки простому та лаконічному синтаксису, великій кількості готових бібліотек і можливості швидкого прототипування рішень. У результаті розробник може за короткий час реалізувати функціональність, протестувати її та внести необхідні зміни без значної витрати часу і ресурсів.

6. Високорівнева мова. Комп'ютер (процесор) безпосередньо розуміє лише обмежений набір елементарних машинних команд. Тому будь-яка програма, написана мовою програмування, фактично є набором інструкцій для виконання певних дій, які необхідно попередньо перетворити на зрозумілий для машини вигляд. Залежно від рівня абстракції розрізняють низькорівневі та високорівневі мови програмування. Якщо програмний код максимально наближений до машинних команд і фактично ними є, такі мови належать до низькорівневих. Якщо ж мова дозволяє писати більш зрозумілий для людини код, який потребує подальшого автоматичного перетворення у машинні інструкції, вона належить до високорівневих мов програмування. Перетворення коду на машинні команди здійснюється спеціальним програмним забезпеченням – компіляторами або інтерпретаторами, які забезпечують виконання програм на комп'ютері.

7. Інтерпретованість. Це означає, що виконання програмного коду здійснюється без попередньої повної компіляції у машинний код – рядки програми обробляються та виконуються інтерпретатором послідовно під час запуску. Такий підхід забезпечує швидке тестування та налагодження програм, оскільки розробник може оперативно виявляти й виправляти помилки без тривалого етапу компіляції. Крім того, інтерпретованість підвищує гнучкість розробки та сприяє швидкому створенню й модифікації програмних рішень.

8. Портативність. Можна запускати програми на різних ОС (*Linux*, *Windows*, *macOS*) без змін вихідного коду (якщо в ньому немає системно-залежних функцій).

9. Динамічна типізація. Це означає, що тип змінної визначається автоматично під час виконання програми і розробнику не потрібно задавати його при оголошенні. Завдяки цьому спрощується процес написання коду, зменшується його обсяг та підвищується гнучкість програмування. Це дозволяє швидше створювати та змінювати програмні рішення,

зосереджуючись на логіці завдання, а не на формальних обмеженнях типізації.

10. Мова загального призначення. Мова призначена для вирішення досить широкого спектру завдань.

11. Мультипарадигмальна мова. *Python* базується на декількох парадигмах програмування: об'єктно-орієнтовна, імперативна, функціональна (функційна), структурна, проте *Python* зазвичай називають об'єктно-орієнтовною мовою програмування. В об'єктно-орієнтованих мовах програмування програми будуються на основі об'єктів, які об'єднують дані і функціонал. Об'єктів у мові *Python* багато, можна навіть вважати, що об'єктами є все (що не дуже коректно, але допустимо).

12. Велика стандартна бібліотека. Стандартна бібліотека містить набір готових модулів і функцій для вирішення типових програмних завдань. Окрім стандартної бібліотеки, екосистема *Python* містить багато зовнішніх високоякісних бібліотек і фреймворків, які доступні через офіційний каталог пакетів *Python* (*PyPI*), що ще більше розширює можливості мови та сфери її застосування.

Попри значну кількість переваг і широкі можливості застосування, мова *Python*, як і будь-який інший інструмент програмування, має певні особливості та обмеження, які необхідно враховувати під час її використання в різних прикладних завданнях:

1. Відносно низька швидкодія порівняно з компільованими мовами програмування. Це зумовлено його інтерпретованим походженням, оскільки код програми виконується через його проміжне перетворення на машинні інструкції безпосередньо під час роботи програми, що є додатковою витратою обчислювальних ресурсів і часу виконання.

2. У *Python* неможливо модифікувати вбудовані класи, зокрема `int`, `str`, `float`, `list` та інші, однак це дозволяє *Python* споживати менше оперативної пам'яті і швидше працювати.

3. Глобальне блокування інтерпретатора *GIL (Global Interpreter Lock)* або проблеми з багатопоточністю. В кожен момент часу може виконуватися тільки одна нитка (потік) коду *Python*, навіть якщо на комп'ютері є кілька процесорів або процесорних ядер. Це зроблено для 12-го унеможливлення руйнування даних при спільній модифікації з різних потоків, оскільки в своїй роботі інтерпретатор *Python* постійно використовує багато ниткових небезпечних даних.

4. Недостатньо досконала підтримка окремих спеціалізованих областей застосування. У деяких випадках це виявляється у відсутності оптимізованих рішень для вузькоспеціалізованих завдань або в обмеженнях продуктивності при роботі з ресурсомісткими системами, що може потребувати використання додаткових інструментів або мов програмування.

5. Підвищене споживання оперативної пам'яті. Це пов'язано з особливостями реалізації мови, зокрема динамічною типізацією та необхідністю зберігання додаткової службової інформації для об'єктів під час виконання програми. У результаті при обробці великих обсягів даних або виконанні завдань вимоги до пам'яті можуть бути вищими порівняно з деякими іншими мовами програмування.

Таким чином, мова *Python* є сучасним, універсальним і широко застосовуваним інструментом програмування, який поєднує простоту вивчення, високу продуктивність розробки, інтерпретований характер, динамічну типізацію, наявність великої стандартної бібліотеки та розвинену екосистему зовнішніх пакетів. Водночас, як і будь-який інший інструмент, *Python* має певні обмеження та особливості, які необхідно враховувати під час його використання в різних прикладних завданнях.

1 БАЗОВІ ПОНЯТТЯ МОВИ PYTHON

1.1 Основні визначення

Програмування – це цілеспрямований процес розробки, формалізації та реалізації алгоритмів у вигляді програм, призначених для виконання на комп'ютерних системах, який включає проєктування, кодування, тестування та супровід програмного забезпечення.

Програма – впорядкована послідовність інструкцій, призначених для виконання на комп'ютері.

Алгоритм – це скінченна, впорядкована та формалізована послідовність однозначно визначених інструкцій, призначених для вирішення певного завдання, виконання яких забезпечує отримання результату за скінченну кількість кроків і може бути реалізоване засобами обчислювальної техніки.

Мова програмування – це формалізована система запису алгоритмів у вигляді, придатному для виконання на комп'ютері.

Інтерпретованість мови програмування – це властивість мови, за якої виконання програмного коду здійснюється без попереднього повного перетворення на машинний код, шляхом його послідовного аналізу та виконання спеціальною програмою-інтерпретатором під час роботи програми.

Інтерпретатор – це спеціальна програмна система, яка здійснює поетапне виконання програмного коду шляхом його безпосереднього аналізу та перетворення на машинні інструкції під час роботи програми без попередньої повної компіляції.

Компіляція – це процес перетворення програмного коду, написаного мовою програмування, на машинний код або проміжне представлення, що виконується комп'ютером, який здійснюється спеціальною програмою, компілятором, до початку виконання програми.

Високорівнева мова програмування – це мова програмування, синтаксис і конструкції якої зорієнтовані на зручність сприйняття людиною та опис алгоритмів у наближеній до природної мови формі, що потребує подальшого перетворення на машинний код за допомогою компілятора або інтерпретатора.

Низькорівнева мова програмування – це мова програмування, команди якої максимально наближені до машинних інструкцій комп'ютера та безпосередньо враховують його апаратні особливості, що забезпечує високу ефективність виконання програм, але ускладнює їх розробку та сприйняття людиною.



Рисунок 1.1 – Порівняльна характеристика високорівневої та низькорівневої мов програмування

Парадигма програмування – це сукупність фундаментальних принципів, підходів і стилів побудови програм, які визначають спосіб організації коду, моделювання завдання та взаємодії між елементами програми. Вона задає загальну концепцію того, як формулюються та

реалізуються алгоритми в межах певної мови програмування або підходу до розробки програмного забезпечення.

Об'єктно-орієнтоване програмування (далі – ООП) – це парадигма програмування, яка базується на представленні програмного забезпечення у вигляді сукупності об'єктів, що поєднують дані (властивості) та методи їх обробки для виконання визначених завдань.

Динамічна типізація – це властивість мов програмування, за якої приналежність значення до певного типу визначається під час виконання програми (*runtime*), а не на етапі компіляції.

Статична типізація – це властивість мов програмування, за якої приналежність значень і змінних до певного типу визначається на етапі компіляції програми, тобто до її виконання, що дозволяє виявляти частину помилок ще під час розробки.



Рисунок 1.2 – Порівняльна характеристика динамічної та статичної типізацій

Імперативне програмування – це парадигма програмування, в якій програма описується як послідовність інструкцій (команд), що змінюють стан програми шляхом поетапного виконання операцій, а основна увага приділяється тому, як саме досягається результат.

Декларативне програмування – це парадигма програмування, в якій описується бажаний результат або властивості вирішення завдання без визначення послідовності дій для його досягнення, тобто акцент робиться на тому, що саме потрібно отримати, а не як саме це зробити.



Рисунок 1.3 – Порівняльна характеристика імперативного та декларативного програмування

Синтаксис мови програмування – це сукупність правил і норм, що визначають допустимі способи запису програмного коду, зокрема структуру команд, порядок їх розташування та використання символів і конструкцій мови.

Семантика – це сукупність правил, що визначають зміст і значення синтаксично коректних конструкцій мови, тобто те, як інтерпретуються та виконуються інструкції програмного коду комп'ютером.

Баг (помилка) – це некоректність у програмному коді або логіці програми, яка призводить до неправильного виконання програми, отримання хибного результату або аварійного завершення її роботи.

1.2 Встановлення *Python*

Python є кросплатформною мовою програмування, що означає її здатність працювати на всіх основних операційних системах. Будь-яка програма, написана мовою *Python*, може виконуватися на сучасному комп'ютері за умови встановлення відповідного інтерпретатора. Водночас процес встановлення та налаштування середовища розробки може незначно відрізнятися залежно від операційної системи.

Спочатку перевіряється наявність актуальної версії *Python* у системі, а в разі її відсутності виконується встановлення. Додатково встановлюється текстовий редактор або середовище розробки. Саме ці два етапи можуть відрізнятися для різних операційних систем і є ключовими для налаштування середовища програмування.

Першим кроком є перевірка наявності встановленого інтерпретатора *Python* у системі. Для цього необхідно відкрити командний рядок (термінал), скориставшись меню «Пуск» або виконавши відповідну команду запуску консолі (натиснути кнопку «Пуск» і в рядку пошуку ввести *cmd* чи «командний рядок», або натиснути комбінацію клавіш *Win + R*, у вікні «Виконати» ввести *cmd* і натиснути на клавішу *Enter*). У відкритому вікні терміналу потрібно ввести команду *py* або *python* у нижньому регістрі та натиснути *Enter*. Якщо у відповідь з'являється інтерактивний режим *Python* (позначений символами `>>>`), то це означає, що *Python* успішно встановлено та може використовуватися. У випадку появи повідомлення про помилку, яке вказує на те, що команда *Python* не розпізнана, це свідчить про відсутність встановленого інтерпретатора або некоректне налаштування системи.

Якщо інтерпретатор *Python* не встановлено, необхідно завантажити інсталяційний файл для операційної системи *Windows*. Для цього потрібно перейти на офіційний сайт (URL: <https://python.org/>) та відкрити розділ *Downloads* («Завантаження»).

На сторінці «Завантаження» буде доступна кнопка для отримання останньої версії *Python*. Після натискання цієї кнопки автоматично розпочнеться завантаження інсталятора, що відповідає вашій операційній системі.

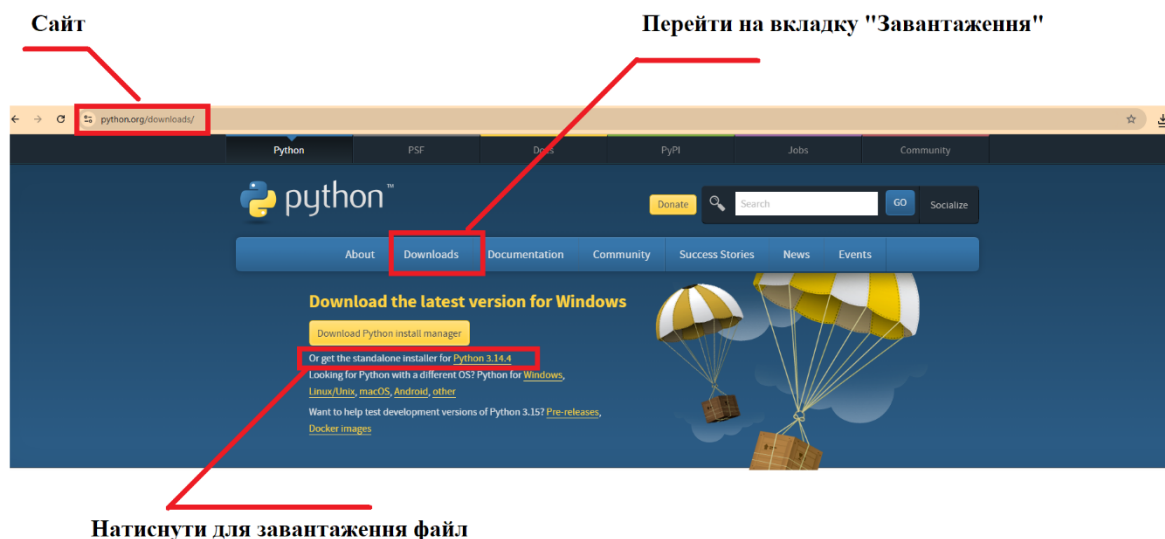


Рисунок 1.4 – Кроки для завантаження інсталятора *Python* з офіційного сайту

Після завершення завантаження необхідно запустити інсталяційний файл та виконати встановлення *Python* (натиснути на кнопку *Install Now*). Під час інсталяції обов'язково потрібно активувати опцію *Add Python to PATH*, яка забезпечує коректну інтеграцію інтерпретатора в систему.



Рисунок 1.5 – Вікно майстра встановлення *Python*

2 АЛГОРИТМІЧНІ СТРУКТУРИ

2.1 Поняття алгоритму та його фундаментальні властивості

Вивчення сучасного програмування неможливе без розуміння фундаментального поняття алгоритму, який є науковою та логічною основою будь-якого обчислювального процесу. В академічному розумінні алгоритм – це впорядкована та формалізована послідовність однозначно визначених інструкцій, спрямованих на вирішення конкретного завдання. У фундаментальному сенсі він є формалізованим описом процедури перетворення вхідних даних (аргументів) на вихідні дані (очікуваний результат). Ключовою ознакою алгоритму є те, що його виконання гарантовано приводить до завершення процесу за обмежену кількість кроків.

Ефективність будь-якого алгоритму визначається його базовими властивостями, а саме:

1. Дискретність – це властивість алгоритму, що передбачає його поділ на послідовність окремих, завершених етапів (кроків). Кожен наступний крок виконується лише після повного завершення попереднього, до того ж результати, отримані на попередньому етапі, стають вхідними даними для виконання наступного.

2. Детермінованість (визначеність) – це властивість, що гарантує однозначність кожної дії алгоритму. Вона передбачає, що після завершення будь-якого кроку система повинна мати чітку вказівку: або перейти до конкретного наступного кроку, або припинити роботу. Завдяки детермінованості алгоритм стає передбачуваним: у разі використання однакових вхідних даних він завжди видаватиме ідентичний результат.

3. Масовість – це властивість, що визначає універсальність алгоритму. Вона передбачає, що алгоритм повинен бути придатним для

вирішення не одного конкретного завдання, а цілого класу однотипних завдань при різних вихідних даних.

4. Результативність (завершеність) – це властивість, яка гарантує, що процес виконання алгоритму не може бути нескінченним. Кожен алгоритм повинен припинятися після виконання певної кількості операцій, надаючи користувачу конкретний результат або обґрунтоване повідомлення про причини, внаслідок яких рішення не може бути отримане).

5. Зрозумілість (формальність) – це властивість, яка означає, що всі команди алгоритму мають бути викладені мовою, зрозумілою конкретному виконавцю (людині чи комп'ютеру). Кожна інструкція повинна бути настільки однозначною та повною, що її можна було б виконати механічно, у суворій послідовності та без будь-яких додаткових пояснень або творчого підходу.

Згідно з класичною теорією програмування будь-який алгоритм незалежно від його складності можна представити за допомогою комбінації трьох базових структур: наслідування (лінійний алгоритм), розгалуження та повторення. Ці структури є універсальними інструментами керування потоком виконання коду, що дозволяють програмі реагувати на зовнішні дані та автоматизувати рутинні операції.

Лінійним називається такий тип алгоритму, який при будь-яких допустимих вхідних даних завдання передбачає одноразове виконання однієї і тієї самої послідовності дій. Це фундаментальна форма організації обчислень, де команди реалізуються суворо послідовно у порядку їхнього запису, без будь-яких умов чи повторів. Кожен крок у такому ланцюгу є обов'язковим, а результат роботи попередньої інструкції зазвичай слугує вихідним пунктом для наступної, що забезпечує чітку логічну неперервність процесу. Базова структура лінійного алгоритму – слідування – характеризується логічною замкненістю. Це означає, що конструкція має лише один, чітко визначений вхід і один вихід; будь-які інші можливості увійти всередину алгоритмічного ланцюга або вийти з нього до завершення

всіх операцій повністю виключені. Така архітектура гарантує однозначність обчислювального шляху: програма проходить крізь усі блоки процесу – від початкового термінала до кінцевого, не відхиляючись від заданого маршруту.

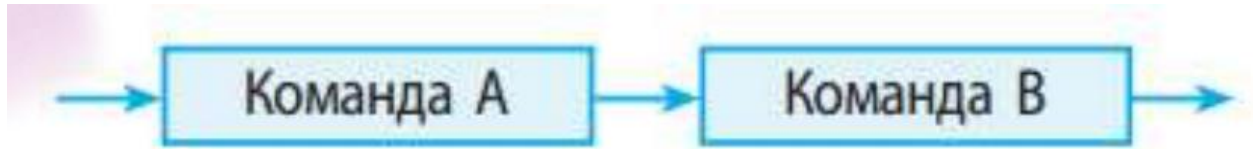


Рисунок 2.1 – Графічне представлення лінійного алгоритму

У практичному програмуванні, зокрема в *Python*, лінійна структура зазвичай використовується для реалізації обчислювальних формул та послідовного введення-виведення даних. Попри свою простоту, наслідування є критично важливим елементом, оскільки навіть у складних об'єктно-орієнтованих системах внутрішня логіка окремих методів зазвичай будується саме за лінійним принципом.

```
# 1. Вхідні дані (Аргументи)
length = float(input("Введіть довжину: "))
width = float(input("Введіть ширину: "))

# 2. Обчислювальний процес (Дія)
area = length * width

# 3. Вихідні дані (Результат)
print(f"Площа прямокутника: {area}")
```

Рисунок 2.2 – Програмна реалізація лінійного алгоритму

Розгалуженим називається такий тип алгоритму, у якому залежно від істинності певної умови (логічного виразу) обирається один із декількох можливих шляхів виконання дій. На відміну від лінійної структури, розгалуження дозволяє програмі адаптуватися до змінних вхідних даних, що робить обчислювальний процес гнучким та варіативним. Кожен такий шлях називається гілкою алгоритму і в конкретному випадку виконується лише

одна з них, після чого потік керування повертається до загальної послідовності. Базова структура розгалуження характеризується наявністю логічного вузла (блоку перевірки умови), який має один вхід і два або більше виходів. Це забезпечує альтернативність обчислювального шляху: програма аналізує вхідні аргументи та залежно від результату («так» або «ні») переходить до відповідного блоку інструкцій. Попри складність внутрішньої логіки, структура розгалуження також залишається замкненою в межах загального алгоритму – незалежно від обраного шляху програма зрештою приходить до єдиної точки завершення цієї конструкції.

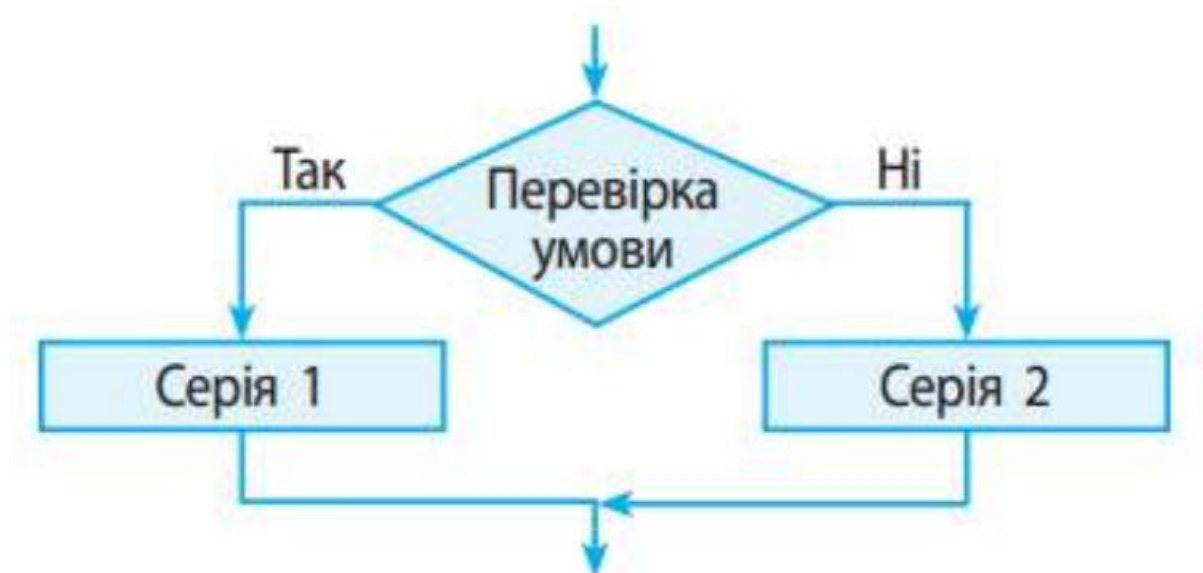


Рисунок 2.3 – Графічне представлення алгоритму розгалуження

У практичному програмуванні розгалуження реалізується за допомогою операторів умовних конструкцій. Це дозволяє створювати складні логічні системи – від простої перевірки пароля до комплексного аналізу технічних параметрів. У межах об’єктно-орієнтованого програмування розгалуження є критично важливим для реалізації методів класів, що повинні виконувати різні дії залежно від стану об’єкта чи отриманих повідомлень. Правильне проєктування умовних конструкцій

забезпечує стійкість програми до некоректних даних та дозволяє реалізувати складні сценарії поведінки системи.

```
# Введення даних
number = float(input("Введіть число для перевірки: "))

# Алгоритм із розгалуженням
if number > 0:
    print("число додатне")
elif number < 0:
    print("число від'ємне")
else:
    print("Ви ввели нуль")
```

Рисунок 2.4 – Програмна реалізація алгоритму розгалуження

Циклічним, або алгоритмом повторення, називається такий тип алгоритму, у якому певна послідовність дій виконується багаторазово доти, доки виконується задана логічна умова. На відміну від лінійних чи розгалужених структур, цикли дозволяють описувати процеси опрацювання великих масивів даних або виконання рутинних операцій за допомогою мінімальної кількості коду. Кожне одноразове виконання дій у циклі називається ітерацією, а сукупність цих дій – тілом циклу. Базова структура циклу характеризується наявністю керівного елемента, який перевіряє умову продовження або завершення повторів. Попри потенційно велику кількість ітерацій, циклічна структура залишається логічно замкненою: вона має один вхід і один вихід. Потік керування утримується всередині конструкції, доки умова істинна, а після її порушення програма обов'язково переходить до наступної команди, що гарантує цілісність загального алгоритму.

У практичному програмуванні алгоритми повторення реалізуються за допомогою спеціальних операторів. Це дозволяє ефективно вирішувати завдання автоматизації – від простого перебору елементів списку до складних обчислювальних процесів у технічному моделюванні. У межах об'єктно-орієнтованого програмування цикли є незамінними для масового

опрацювання об'єктів, оновлення станів систем у реальному часі та взаємодії з динамічними структурами даних.

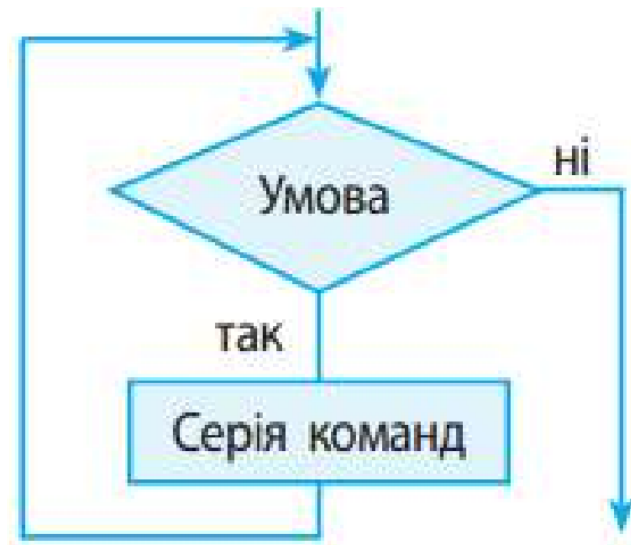


Рисунок 2.5 – Графічне представлення циклічного алгоритму

```
# Список значень (наприклад, потужність ламп у Вт)
powers = [40, 60, 100, 150]

print("Розрахунок споживання енергії за годину:")

# Циклічний алгоритм (повторення дії для кожного елемента)
for p in powers:
    energy = p * 1 # Потужність * 1 година
    print(f"лампа {p}Вт споживає {energy} Вт*год")
```

Рисунок 2.6 – Програмна реалізація циклічного алгоритму

2.2 Умовні оператори

Умовний оператор, або умовна інструкція, – це інструмент програмування, який дозволяє виконувати різні блоки програмного коду залежно від істинності або хибності певної умови. Фактично, умовний оператор створює в програмі «точку розгалуження», де на підставі аналізу даних приймається рішення про подальший шлях обчислювального процесу.

Загальна схема функціонування умовного оператора будується за такими етапами:

- перевірка логічного виразу: насамперед аналізується певна умова, значення якої інтерпретується як істина (*True*) або хибне значення (*False*);
- виконання основної гілки: якщо вираз-умова є істиною, програма реалізує першу виокремлену послідовність команд;
- виконання альтернативної гілки: якщо умова інтерпретується як хибна, виконується інша (альтернативна) послідовність інструкцій;
- замкненість структури: після завершення обраної гілки команд керування автоматично передається наступній інструкції, яка міститься безпосередньо за блоком умовного оператора.

У середовищі *Python* така логіка реалізується за допомогою ключових слів:

1) *if* («якщо») – це базова інструкція розгалуженого алгоритму, яка забезпечує перевірку істинності певного логічного виразу. Вона виконує прикріплений до неї блок програмного коду лише в тому випадку, коли результат перевірки умови інтерпретується як істинний (*True*);

```
# Перевірка наявності знижки
price = 1000
is_member = True

if is_member:
    price = price * 0.9 # Знижка 10%
    print("Знижка активована!")

print(f"Сума до сплати: {price}")
```

Рисунок 2.7 – Програмна реалізація конструкції *if*

2) *else* («інакше») – це інструкція умовного розгалуження, яка визначає блок коду, що має бути виконаний у випадку, коли основна умова (*if*) виявилася хибною (*False*). Він є запасним варіантом, гарантуючи, що

програма виконає певну дію навіть тоді, коли жодна з попередньо заданих позитивних умов не спрацювала. Використання *else* перетворює неповне розгалуження на повне, забезпечуючи альтернативний шлях для потоку керування;

```
# Перевірка доступу до системи
age = int(input("Введіть ваш вік: "))

if age >= 18:
    print("Доступ дозволено. Ласкаво просимо!")
else:
    print("Доступ обмежено. Ви занадто юні.")
```

Рисунок 2.8 – Програмна реалізація конструкції *if...else*

3) *elif* – це конструкція, що дозволяє реалізувати вибір одного з багатьох можливих шляхів виконання алгоритму. Вона використовується для перевірки додаткових умов, якщо попередня умова *if* була хибною. Завдяки *elif* створюється ланцюжок перевірок: програма послідовно аналізує кожен логічний вираз до першої істинної умови, виконує відповідний блок коду та завершує роботу всієї конструкції, що робить обчислювальний процес гнучким та багатоваріантним;

```
# Визначення категорії товару за ціною
item_price = 1500

if item_price < 500:
    category = "Бюджетний"
elif item_price <= 2000:
    category = "Середній сегмент"
else:
    category = "Преміум"

print(f"Товар належить до категорії: {category}")
```

Рисунок 2.9 – Програмна реалізація конструкції *if...elif...else*

Для того щоб алгоритми розгалуження працювали коректно, необхідно суворо дотримуватися правил синтаксису *Python*. Ключовою особливістю є те, що після ключових слів *if* та *elif* обов'язково повинна бути прописана умова (логічний вираз). Без неї інструкція втрачає сенс, оскільки не буде критерію для вибору шляху виконання. Натомість після ключового слова *else* умова ніколи не пишеться, оскільки цей блок спрацьовує автоматично, як варіант за замовчуванням, якщо всі попередні перевірки виявилися хибними. Кожна секція розгалуження обов'язково завершується двокрапкою «:», яка вказує інтерпретатору на початок вкладеного блоку команд.

```
if умова_1:           # Обов'язкова умова №1
    команда_1         # Виконується, якщо умова_1 == True
elif умова_2:         # Додаткова умова (якщо перша False)
    команда_2         # Виконується, якщо умова_2 == True
else:                 # Умова не пишеться (в усіх інших випадках)
    команда_3         # Виконується, якщо всі умови вище == False
```

Рисунок 2.10 – Логіка написання конструкцій *if...elif...else*

Важливою вимогою щодо оформлення коду в *Python* є система відступів. Усі інструкції, що належать до певної гілки розгалуження, повинні мати однаковий відступ – за стандартом це 4 пробіли. Саме відступи визначають ієрархію коду та вказують на те, які саме команди виконуються при істинності конкретної умови. Самі умови зазвичай будуються за допомогою знаків порівняння: `==` (рівно), `!=` (не дорівнює), `>`, `<`, `>=`, `<=`. Між ключовим словом та логічним виразом обов'язково ставиться пробіл для читабельності та коректної роботи інтерпретатора. Правильне проєктування таких конструкцій забезпечує стійкість програми щодо помилок та дозволяє реалізувати складні сценарії поведінки системи залежно від вхідних параметрів.

```

# Приклад: Автоматизована система контролю температури
temperature = 25 # Вхідний параметр

if temperature > 30:
    # Відступ 4 пробіли – цей блок належить до умови 'if'
    status = "Критичний рівень"
    print("Увага! Активувати систему охолодження.")
    print(f"Поточний статус: {status}")

elif temperature == 25:
    # Пробіл після 'elif', використання '==' та відступи
    status = "Нормальний рівень"
    print("Температура в межах норми.")
    print(f"Поточний статус: {status}")

else:
    # Блок 'else' без умови, але з обов'язковим відступом
    status = "Знижений рівень"
    print("Температура нижче норми. Увімкнути обігрів.")

# Код без відступу – виконується завжди після завершення розгалуження
print("Моніторинг завершено успішно.")

```

Рисунок 2.11 – Приклад практичної реалізації умовної конструкції з використанням відступів та операторів порівняння

2.3 Циклічні оператори

У процесі вирішення завдань зазвичай виникає необхідність багаторазового виконання певної дії або послідовності операцій. Для автоматизації таких процесів у програмуванні використовують конструкцію повторення або цикл. Кожне окреме виконання дій у циклі називається його кроком, або ітерацією.

Будь-яка циклічна конструкція складається з двох основних частин – заголовка та тіла циклу. Заголовок відповідає за логіку керування повторами, тобто визначає умову, за якої цикл продовжує свою роботу або завершується. Тіло циклу містить безпосередній набір команд, які мають виконуватися на кожній ітерації.

У мові програмування *Python* для реалізації циклічних алгоритмів використовуються два основні види операторів, які розрізняються за способом керування повтореннями (цикл із передумовою та ітераційний цикл):

1. Оператор циклу з передумовою (*while*) – це інструкція, що забезпечує багаторазове виконання блоку коду, доки заданий логічний вираз залишається істинним (*True*). Принцип роботи цього оператора такий: щойно черга доходить до його виконання, аналізується умова, вказана після ключового слова *while*. Якщо вона істинна, виконуються команди в тілі циклу, після чого система знову повертається до заголовка для перевірки. Цей процес повторюється доти, доки під час чергової перевірки вираз не виявиться хибним. На цьому виконання циклу завершується, а керування передається команді, яка йде безпосередньо за цією конструкцією.

```
# Початкова сума грошей
balance = 100
price = 35

print(f"Початковий баланс: {balance} грн. Ціна морозива: {price} грн.")

# Цикл з передумовою (заголовок циклу)
while balance >= price:
    # Тіло циклу (дії, що повторюються)
    balance = balance - price
    print(f"Куплено морозиво. Залишок: {balance} грн.")

# Команда після завершення циклу
print("Грошей на наступну покупку недостатньо. Цикл завершено.")
```

Рисунок 2.12 – Програмна реалізація оператора *while*

Окрім основного тіла циклу, конструкція *while* може містити додатковий блок *else*. Команди, вказані після ключового слова *else*, виконуються лише тоді, коли умова в заголовку циклу стає хибною (*False*). Це відбувається або на самому початку (якщо умова не виконана жодного

разу), або після успішного завершення всіх ітерацій циклу. Блок *else* виконується тільки один раз і лише на завершальній стадії роботи оператора.

```
# Лічильник
count = 1

while count <= 3:
    print(f"Крок №{count}")
    count += 1
else:
    # Виконається після того, як умова count <= 3 стане хибною
    print("Умова більше не виконується, переходимо до else.")

print("Програма завершена.")
```

Рисунок 2.13 – Базовий приклад циклу *while* з блоком *else*

Варто зауважити, що аналогічного результату можна досягти, розмістивши команди після циклу. Проте практична цінність блоку *else* зростає при використанні спеціальних інструкцій керування потоком:

- *break* («переривання») – ця інструкція негайно зупиняє роботу циклу. Важливою особливістю є те, що при використанні *break* блок *else* не виконується. Це дозволяє розрізнити ситуації, коли цикл завершився (умова стала хибною) та коли він був перерваний примусово (наприклад, через виявлення помилки або знаходження потрібного значення).

```
count = 1

while count <= 5:
    if count == 3:
        print("Зустріли число 3, перериваємо цикл!")
        break # Вихід із циклу, блок else не спрацює
    print(f"Крок №{count}")
    count += 1
else:
    print("Цикл завершено природним шляхом.")

print("Програма завершена.")
```

Рисунок 2.14 – Базовий приклад циклу *while...else* з інструкцією *break*

Логіка роботи цього прикладу полягає в послідовному виконанні ітерацій, під час яких цикл встигає вивести повідомлення «Крок № 1» та «Крок № 2». Однак щойно значення змінної *count* досягає 3, спрацьовує інструкція *break*, яка спричиняє миттєвий вихід із циклу. Важливою особливістю такого переривання є те, що блок *else* повністю ігнорується, тому текст «Цикл завершено природним шляхом» не буде виведений на екран, а програма одразу перейде до виконання команд, розташованих поза межами циклічної конструкції:

- *continue* («продовження») – ця інструкція завершує виконання лише поточної ітерації та здійснює миттєвий перехід до перевірки умови в заголовку *while*. Вона дозволяє пропустити решту команд у тілі циклу, не завершуючи сам цикл та не впливаючи на фінальне виконання блоку *else*.

```
count = 0

while count < 5:
    count += 1
    if count == 3:
        print("Число 3 пропускаємо...")
        continue # Повертаємось на початок циклу, ігноруючи наступний print
    print(f"Крок №{count}")
else:
    print("Цикл завершено повністю (else спрацював).")

print("Програма завершена.")
```

Рисунок 2.15 – Базовий приклад циклу *while...else* з інструкцією *continue*

Логіка роботи цього прикладу базується на тому, що коли значення змінної *count* дорівнює 3, спрацьовує інструкція *continue*. У цей момент програма ігнорує виконання команди виведення тексту для поточного кроку і одразу повертається до заголовка циклу для перевірки умови *while*. У результаті в консолі відобразяться кроки 1, 2, 4 та 5, тоді як крок 3 буде пропущений. Ключова різниця з оператором *break* полягає в тому, що цикл повністю не переривається, а лише пропускається конкретна ітерація, тому блок *else* обов’язково виконається у самому кінці, коли умова *count < 5* стане хибною.

3 ФУНКЦІЇ

3.1 Загальна інформація про функції в *Python*

Під час розроблення програмного забезпечення зазвичай виникає необхідність багаторазового виконання однакових або подібних послідовностей команд. Повторення одних і тих самих фрагментів коду в різних частинах програми ускладнює її розробку, знижує читабельність та ускладнює подальший супровід. Для вирішення цієї проблеми в сучасних мовах програмування використовуються функції, які дозволяють об'єднувати логічно пов'язані інструкції в окремі іменовані блоки коду.

Функції є одним із фундаментальних засобів структуризації програм і реалізують принцип модульності програмування. Вони дають змогу розбивати складні завдання на менші підзавдання, повторно використовувати програмний код та спрощувати процес його тестування і налагодження. Завдяки цьому програми стають більш зрозумілими, гнучкими та зручними для подальшого розвитку.

У мові *Python* функція становить іменований блок програмного коду, який виконує певне завдання та може бути викликаний з будь-якого місця програми необмежену кількість разів. Використання функцій є одним із основних підходів до організації програмного коду та важливим елементом професійної розробки програмного забезпечення.

Вони поділяються на вбудовані та користувацькі (власні). Вбудовані функції входять до складу мови *Python* та доступні для використання без додаткового оголошення. До них належать, наприклад, функції для виведення даних на екран *print()*, визначення довжини об'єкта *len()* або отримання типу даних *type()* тощо. Користувацькі функції створюються програмістом для реалізації власної логіки та багаторазового використання певного фрагмента коду.

Окремим різновидом є анонімні або *lambda*-функції. Вони не мають власного імені та використовуються для запису простих обчислень у компактній формі. Такі функції зазвичай застосовуються у випадках, коли невелика функція потрібна лише один раз або передається як аргумент іншій функції.

3.2 Синтаксис оголошення та виклику функцій

Під час роботи з функціями необхідно розрізняти два основні етапи: визначення (оголошення) функції та її виклик. Визначення функції полягає у створенні іменованого блоку коду, який містить певну послідовність інструкцій для вирішення конкретного завдання. Після визначення функція стає доступною для використання в програмі.

Виклик функції – це процес передачі керування до її програмного коду з метою виконання передбачених у ній дій. Одну й ту саму функцію можна викликати необмежену кількість разів із різними вхідними даними, що забезпечує повторне використання коду та сприяє підвищенню ефективності розроблення програмного забезпечення.

Будь-яка функція в мові *Python* складається з двох основних частин: заголовка та тіла функції. Заголовок функції використовується для її оголошення і містить інформацію про ім'я функції та перелік параметрів, необхідних для її роботи. Тіло функції становить набір інструкцій, які будуть виконуватися під час виклику функції.

У *Python* заголовок і тіло функції оформлюються як основна та вкладена інструкції. Такий підхід ґрунтується на використанні відступів, які є важливою складовою синтаксису мови. Саме відступи визначають межі тіла функції та об'єднують операторів у єдиний логічний блок.

Оголошення функції починається зі службового слова *def*, після якого вказується ім'я функції. Далі в круглих дужках може розміщуватися список

формальних параметрів, через які до функції передаються вхідні дані. Заголовок функції завершується символом двокрапки «:», який вказує на початок блоку інструкцій, що належать до тіла функції.

```
# Оголошення (визначення) функції
def greet_user(username): # def - ключове слово; greet_user - ім'я функції
    # username - це параметр функції (змінна, що приймає вхідні дані)

    # Тіло функції (вкладений блок команд із обов'язковим відступом у 4 пробіли)
    print(f"Вітаємо, {username}! Раді бачити вас у системі.")

# Виклик функції (звернення до коду за його іменем)
greet_user("Олексій") # "Олексій" - це аргумент (фактичне значення), що передається
greet_user("Марія")   # Повторний виклик функції з іншим аргументом
```

Рисунок 3.1 – Синтаксична структура та приклад оголошення функції

З нового рядка записується тіло функції. Воно може містити одну або декілька інструкцій, які виконуються послідовно під час виклику функції. Усі оператори тіла функції повинні мати однаковий відступ відносно заголовка, що є обов'язковою вимогою синтаксису *Python*.

Ім'я функції є ідентифікатором, який формується відповідно до правил іменування в *Python* та використовується для подальшого виклику функції в програмі. Для підвищення читабельності коду рекомендується обирати змістовні імена, які відображають призначення функції.

Під час створення імен функцій необхідно дотримуватися встановлених правил синтаксису мови *Python*. Ім'я функції може містити літери латинського алфавіту, цифри та символ підкреслення. Водночас ім'я не може починатися з цифри та не повинно збігатися із зарезервованими словами мови програмування. Крім того, *Python* є чутливим до регістру символів, тому імена *myfunc*, *MyFunc* та *MYFUNC* розглядаються як різні ідентифікатори.

Після оголошення функції її можна використовувати в програмі шляхом виклику. Функція викликається за її ім'ям із зазначенням круглих дужок, у яких за потреби передаються необхідні аргументи.

Під час виклику функції значення аргументів передаються параметрам, визначеним у заголовку функції. Після цього виконується тіло функції, а керування повертається до того місця програми, звідки було здійснено виклик.

Якщо функція не має параметрів, її виклик здійснюється лише за допомогою імені та порожніх круглих дужок.

Одну й ту саму функцію можна викликати необмежену кількість разів у різних частинах програми.

Важливо пам'ятати, що перед викликом функція повинна бути визначена в програмі. Спроба викликати функцію до її оголошення призведе до виникнення помилки виконання.

3.3 Параметри та аргументи функцій

У більшості випадків функції використовуються для обробки даних, які передаються їм під час виклику. Для цього в *Python* передбачені параметри і аргументи. Завдяки використанню параметрів одна й та сама функція може виконувати однакові дії з різними наборами даних без зміни її програмного коду.

Параметри функції (формальні) – це змінні, які оголошуються в заголовку функції та використовуються всередині її тіла для виконання необхідних обчислень або інших операцій. Якщо функція потребує декількох параметрів, вони вказуються в круглих дужках через кому.

Під час виклику функції параметрам передаються конкретні значення, які називаються аргументами (фактичні параметри). Таким чином, параметри є змінними, що визначаються під час оголошення функції, а аргументи – це фактичні дані, які передаються функції під час її виклику.

На рисунку 3.2 продемонстрована передача аргументів до функції та використання параметрів під час виконання обчислень. Параметри *voltage* та

current, оголошені в заголовку функції, становлять локальні змінні, доступні лише всередині функції. Вони використовуються для зберігання вхідних даних, необхідних для виконання обчислень і не впливають безпосередньо на змінні, що містяться за межами функції.

Під час виклику функції відбувається передача аргументів, значення яких автоматично присвоюються відповідним параметрам. Наприклад, при виклику *calculate_power(220, 5)* значення 220 передається параметру *voltage*, а значення 5 – параметру *current*. При наступному виклику ці самі параметри можуть отримувати вже інші значення, наприклад 380 та 10.

```
# Оголошення функції з двома параметрами
def calculate_power(voltage, current):
    # voltage та current – це параметри (змінні-пакети для вхідних даних)

    # Обчислення активної потужності: P = U * I
    power = voltage * current
    print(f"При напрузі {voltage} В та струмі {current} А потужність становить: {power}")

# Виклик функції з різними фактичними значеннями (аргументами)
calculate_power(220, 5) # Передаємо аргументи 220 та 5
calculate_power(380, 10) # Передаємо аргументи 380 та 10
```

Рисунок 3.2 – Приклад використання параметрів та аргументів функції у *Python*

На відміну від звичайних змінних, значення параметрів не задаються безпосередньо в тілі функції, а надходять із програми під час її виконання. Після отримання аргументів функція може використовувати їх для виконання необхідних дій, обчислень або формування результату.

Мова *Python* надає широкі можливості для роботи з параметрами функцій, що забезпечує гнучкість під час розроблення програм. Зокрема, функції можуть містити параметри із значеннями за замовчуванням (коли певний параметр набуває одного й того самого значення). Значення за замовчуванням задається безпосередньо під час оголошення функції. Для цього після імені параметра записується оператор присвоєння `=`, а потім

вказується значення, яке буде використовуватися у випадку, якщо під час виклику функції відповідний аргумент не переданий. Якщо користувач не вкаже значення для такого параметра, функція автоматично використає значення за замовчуванням. Водночас за потреби це значення може бути перевизначене шляхом передачі іншого аргументу під час виклику функції.

```
# Оголошення функції (ПДВ за замовчуванням 20%)
def get_price(net_price, vat=0.20):
    total = net_price * (1 + vat)
    print(f"Ціна: {net_price} | ПДВ: {vat} | Разом: {total}")

# 1. Виклик з одним аргументом (ПДВ = 20%)
get_price(5000)

# 2. Виклик з двома аргументами (ПДВ = 7%)
get_price(5000, 0.07)
```

Рисунок 3.3 – Приклад використання параметрів із замовчуванням

Крім того, *Python* підтримує використання іменованих (ключових) аргументів, що дозволяє передавати значення параметрам за їхніми назвами незалежно від порядку розташування.

```
# Оголошення функції з параметрами за замовчуванням
def func(a, b=2, c=10):
    print(f"a={a} b={b} c={c}")

# 1. Виклик із ключовим аргументом c=15 (b береться за замовчуванням)
func(3, c=15)

# 2. Виклик, де всі аргументи після позиційного задані як ключові
func(3, c=15, b=8)
```

Рисунок 3.4 – Приклад використання ключових аргументів

Також у *Python* існує можливість створення функцій, здатних приймати довільну кількість аргументів, що є корисним у випадках, коли кількість вхідних даних заздалегідь невідома. Механізм змінної кількості аргументів

підтримується як для позиційних (це аргументи, значення яких передаються параметрам відповідно до їх порядку розташування у виклику функції), так і для ключових аргументів. Для вказування цього типу аргументів використовується зірочка (*) перед ім'ям параметра у визначенні функції.

```
# Оголошення функції з довільною кількістю аргументів
def calc_total_power(*appliances):
    # appliances – це кортеж, який містить усі передані значення
    total = sum(appliances)
    print(f"Підключено пристроїв: {len(appliances)} | Сумарна потужність: {total} Вт")

# 1. Виклик функції з трьома аргументами
calc_total_power(60, 100, 40)

# 2. Виклик функції з п'ятьма аргументами
calc_total_power(11, 15, 11, 24, 18)
```

Рисунок 3.5 – Приклад використання змінної кількості аргументів

Завдяки таким можливостям параметри та аргументи у *Python* є потужним інструментом для створення універсальних, гнучких та зручних під час використання функцій.

3.4 Простори імен та області видимості

Під час роботи з функціями важливим поняттям є область видимості змінних. Область видимості визначає частину програми, в межах якої можна звертатися до певної змінної. Залежно від місця оголошення змінні поділяються на глобальні та локальні.

Кожна функція у *Python* формує власний простір імен, у межах якого можуть створюватися локальні змінні. Локальною називається змінна, яка оголошена або отримує значення всередині функції. Така змінна доступна лише в межах цієї функції та не може бути використана за її межами. Це дозволяє ізолювати внутрішні дані функції від інших частин програми та запобігти випадковій зміні їх значень.

Змінні, оголошені в основній частині програми поза функціями, називаються глобальними. Вони належать до глобального простору імен і можуть бути доступними з різних частин програми, зокрема й усередині функцій. Використання глобальних змінних надає можливість зберігати дані, які повинні бути доступні для кількох функцій або для всієї програми в цілому.

```
# Глобальна змінна
x = "Глобальний"

def f():
    # Локальна змінна з таким самим ім'ям
    x = "Локальний"
    print(x) # Виведе локальне значення

f()
print(x)    # Виведе глобальне значення
```

Рисунок 3.6 – Приклад застосування локальних та глобальних змінних

Важливо розуміти, що локальні та глобальні змінні з однаковими іменами є різними об'єктами. Наприклад, якщо в основній програмі оголошено змінну X, а всередині функції створено іншу змінну з таким самим ім'ям X, то кожна з них буде зберігати власне значення незалежно одна від одної. Це пояснюється тим, що вони належать до різних просторів імен.

Таким чином, область видимості змінних забезпечує ізоляцію даних між окремими частинами програми, запобігає конфліктам імен та сприяє більш структурованій організації програмного коду.

У деяких випадках виникає необхідність не лише отримати доступ до глобальної змінної всередині функції, а й змінити її значення. За замовчуванням *Python* розглядає будь-яку змінну, якій присвоюється значення всередині функції, як локальну, тому спроба змінити глобальну змінну без додаткових вказівок призведе до створення нової локальної змінної з таким самим ім'ям.

Для того щоб повідомити *Python*, що всередині функції необхідно працювати саме з глобальною змінною, використовується службове слово *global*. Його вказують перед ім'ям змінної на початку тіла функції.

Після оголошення змінної як глобальної всі операції з нею всередині функції будуть виконуватися над змінною, що міститься в глобальному просторі імен. Таким чином, зміни, внесені у функції, зберуться і після завершення її виконання.

```
# Глобальна змінна
status = "Вимкнено"

def activate_system():
    global status # Вказуємо, що будемо змінювати саме глобальну змінну status
    status = "Активно" # Змінюємо її значення

print(f"До виклику: {status}")
activate_system()
print(f"Після виклику: {status}")
```

Рисунок 3.7 – Приклад використання слова *global*

Проте краще не змінювати значень глобальних змінних всередині функції. Якщо за функцією необхідно змінити значення змінної, краще повернути це значення функцією і в головній програмі виконати операцію зміни. Якщо дотримуватися цього правила, то функції будуть незалежними від коду головної програми і їх можна легко копіювати з однієї програми в іншу.

Окрім локальних та глобальних змінних є ще один тип змінних, так звана нелокальна (*nonlocal*) змінна. Нелокальна змінна є чимось середнім між локальною та глобальною. Такі змінні трапляються у функціях, які визначені всередині інших функцій. Для оголошення нелокальної змінної використовується службове слово *nonlocal*.

3.5 Оператор *return*

Під час виконання функції зазвичай виникає необхідність передати результат її роботи до місця виклику. Для цього в мові *Python* використовується оператор *return*.

Оператор *return* завершує виконання функції та повертає вказане значення до тієї частини програми, звідки було здійснено виклик функції. Після виконання оператора *return* усі подальші інструкції в тілі функції ігноруються, а керування передається до програми, що викликала функцію.

Розглянемо приклад функції, яка визначає більше з двох чисел.

```
def maximum(x, y):  
    if x > y:  
        return x  
    elif x == y:  
        return "числа рівні"  
    else:  
        return y  
  
# Обробка та виведення результату  
result = maximum(2, 3)  
print(result) # Виведе: 3
```

Рисунок 3.8 – Приклад використання оператора *return*

У наведеному прикладі функція *maximum()* приймає два параметри та за допомогою умовного оператора *if* визначає найбільше значення. Після цього оператор *return* повертає знайдений результат до місця виклику функції.

Оператор *return* може повертати різні типи даних, зокрема числа, рядки, списки, словники та інші об'єкти *Python*. Також допускається використання оператора *return* без вказування значення. У такому випадку функція повертає спеціальне значення *None*, яке означає відсутність результату.

```
def my_function():  
    return # Порожній return просто завершує роботу функції  
  
# Виклик функції та виведення її результату  
result = my_function()  
print(result) # Виведе: None
```

Рисунок 3.9 – Приклад використання оператора *return* за відсутності вказаного об'єкта повернення

У цьому випадку після виклику функції буде повернуто значення *None*. Варто зазначити, що кожна функція в *Python* неявно містить оператора *return None* наприкінці свого тіла. Тому якщо оператор *return* не вказаний, після завершення роботи функції автоматично повертається значення *None*.

3.6 Анонімні функції: функція *lambda*

Як зазначалося раніше, за допомогою службового слова *def* у *Python* створюються іменовані функції, які після оголошення можуть викликатися з будь-якого місця програми за їхнім ім'ям. Однак у мові *Python* також передбачено спеціальний механізм створення функцій без імені, які називаються анонімними, або *lambda*-функціями.

Lambda-функція становить компактний спосіб опису простої функції та створюється за допомогою ключового слова *lambda*. На відміну від звичайних функцій, *lambda*-функції не мають власного імені та зазвичай складаються з одного виразу. Результат обчислення цього виразу автоматично повертається як результат роботи функції.

```
# Оголошення lambda-функції для обчислення площі прямокутника (S = a * b)  
calc_area = lambda a, b: a * b  
  
# Виклик функції та виведення результату  
result = calc_area(5, 4)  
print(result) # Виведе: 20
```

Рисунок 3.10 – Приклад створення *lambda*-функції

Створену *lambda*-функцію можна присвоїти змінній, після чого до неї можна звертатися так само, як і до звичайної функції. Крім того, анонімні функції зазвичай використовуються як аргументи інших функцій або в ситуаціях, коли невелика функція потрібна лише в одному місці програми.

Застосування *lambda*-функцій дозволяє скоротити обсяг коду та зробити його більш компактним тоді, коли немає потреби створювати окрему іменовану функцію.

4 ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

4.1 Вступ до об'єктно-орієнтованого програмування

У процесі розроблення програмного забезпечення складність програм поступово зростає, що потребує використання підходів, які забезпечують зручну організацію коду, його повторне використання та спрощують подальший супровід програм. Одним із таких підходів є об'єктно-орієнтоване програмування (ООП), яке на сьогодні є однією з найпоширеніших парадигм розроблення програмного забезпечення.

На відміну від процедурного програмування, де основна увага приділяється послідовності виконання команд і функцій, об'єктно-орієнтоване програмування базується на представленні програми у вигляді взаємодіючих об'єктів. Кожен об'єкт поєднує в собі дані та засоби їх обробки, що дозволяє створювати більш структурований, зрозумілий та масштабований програмний код.

Мова *Python* повністю підтримує об'єктно-орієнтований підхід і надає розробнику зручні інструменти для створення класів, об'єктів та реалізації основних принципів ООП. Розуміння цих принципів є важливим етапом

підготовки сучасного програміста та необхідною умовою для розробки складних програмних систем.

Отже, об'єктно-орієнтоване програмування (ООП) – це парадигма програмування, яка базується на представленні програмної системи у вигляді сукупності взаємодіючих об'єктів. Кожен об'єкт поєднує в собі дані, що характеризують його стан, та методи, які визначають його поведінку і дозволяють виконувати певні дії з цими даними.

Ефективність об'єктно-орієнтованого програмування забезпечується низкою фундаментальних принципів, які визначають підходи до проєктування та реалізації програмних систем. Використання цих принципів дозволяє створювати зрозумілий, гнучкий і масштабований програмний код, що легко підтримується та розширюється в процесі його розвитку.

До основних принципів об'єктно-орієнтованого програмування належать інкапсуляція, наслідування, поліморфізм, абстракція.

Кожен із цих принципів виконує важливу роль у побудові програмних систем і сприяє ефективній організації взаємодії між об'єктами.

Інкапсуляція – це один із базових принципів об'єктно-орієнтованого програмування, який полягає в об'єднанні даних (атрибутів) та методів їх обробки в межах одного класу або об'єкта, а також у приховуванні деталей внутрішньої реалізації від зовнішнього середовища.

Основна ідея інкапсуляції полягає в тому, що внутрішній стан об'єкта повинен бути захищений від неконтрольованого доступу та змінювання. Взаємодія з об'єктом здійснюється через спеціально визначені методи, які забезпечують контроль над отриманням і модифікацією даних. Таким чином, користувач або інші об'єкти працюють не безпосередньо з внутрішньою структурою об'єкта, а через його відкритий інтерфейс.

Використання інкапсуляції дозволяє підвищити надійність програмного забезпечення, зменшити ймовірність помилок та забезпечити цілісність даних. Крім того, приховування деталей реалізації спрощує супровід програмного коду, оскільки внутрішня будова об'єкта може

змінюватися без впливу на інші частини програми, які використовують цей об'єкт.

Наслідування – це ще один із основних принципів об'єктно-орієнтованого програмування, який дозволяє створювати нові класи на основі вже існуючих. Новий клас, який називається дочірнім, успадковує атрибути та методи батьківського класу, а також може доповнювати їх новими властивостями або змінювати існуючу поведінку.

Використання наслідування є особливо корисним під час розробки великих програмних систем, у яких різні об'єкти мають спільні характеристики та функціональні можливості. Замість повторного створення однакового коду для кожного нового класу розробник може використати вже реалізовану функціональність батьківського класу та за потреби розширити її.

Такий підхід дозволяє скоротити обсяг програмного коду, уникнути його дублювання та спростити подальший супровід програмного забезпечення. Крім того, наслідування сприяє побудові ієрархії класів, у якій більш спеціалізовані класи отримують властивості та методи від більш загальних класів.

Поліморфізм – це інший принцип об'єктно-орієнтованого програмування, який полягає у здатності різних об'єктів по-різному реагувати на один і той самий виклик методу або виконувати однакові за призначенням дії власним способом. Інакше кажучи, одна й та сама операція може реалізовуватися по-різному залежно від типу об'єкта, до якого вона застосовується.

Поліморфізм тісно пов'язаний із наслідуванням, оскільки дочірні класи можуть перевизначати методи батьківського класу та реалізовувати їх відповідно до власних особливостей. Наприклад, метод «Відобразити інформацію» може існувати в різних класах, але для кожного типу об'єкта він буде виводити різні дані.

Використання поліморфізму дозволяє створювати універсальні алгоритми та функції, які можуть працювати з об'єктами різних класів через єдиний інтерфейс. Це підвищує гнучкість програмного коду, спрощує його розширення та зменшує залежність між окремими компонентами програмної системи.

Абстракція – це принцип, який полягає у виділенні найбільш важливих характеристик об'єкта та приховуванні деталей його внутрішньої реалізації, що не є суттєвими для користувача або інших частин програми.

Головною метою абстракції є спрощення роботи зі складними програмними системами шляхом зосередження уваги на ключових властивостях і поведінці об'єктів. Завдяки цьому розробник може взаємодіяти з об'єктом через зрозумілий інтерфейс, не заглиблюючись у особливості його внутрішньої реалізації.

Абстракція тісно пов'язана з інкапсуляцією та доповнює її. Якщо інкапсуляція забезпечує приховування внутрішніх даних і механізмів роботи об'єкта, то абстракція визначає, які саме властивості та методи повинні бути доступними для використання. Це дозволяє зменшити складність програмного коду та зробити його більш зрозумілим і зручним для супроводу.

Для ефективного використання об'єктно-орієнтованого програмування необхідно розуміти його основні поняття, до яких належать клас, об'єкт, атрибути та методи. Саме вони є основою побудови програмних систем, розроблених відповідно до принципів ООП.

Клас – це користувацький тип даних, який описує структуру та поведінку майбутніх об'єктів. Клас визначає набір атрибутів і методів, якими будуть володіти створені на його основі об'єкти. Інакше кажучи, клас можна розглядати як шаблон або модель, за якою створюються конкретні об'єкти.

Об'єкт – це екземпляр класу, створений на основі його опису. Кожен об'єкт має власний стан, який визначається значеннями його атрибутів і може виконувати дії, передбачені методами відповідного класу.

Атрибути – це змінні, оголошені в класі, які використовуються для зберігання даних та характеризують стан об'єкта. Значення атрибутів можуть відрізнятися для різних об'єктів одного й того самого класу.

Методи – це функції, визначені всередині класу, які описують поведінку об'єктів та реалізують операції з їхніми даними. За допомогою методів об'єкти можуть виконувати певні дії, змінювати власний стан або взаємодіяти з іншими об'єктами.

4.2 Класи, об'єкти та методи

Для створення класу в *Python* використовується службове слово *class*, після якого вказується ім'я класу та символ двокрапки «:». Усередині класу можуть бути визначені атрибути, що зберігають дані об'єкта, а також методи, які реалізують його функціональні можливості.

```
# Оголошення класу, що містить лише атрибути (властивості)
class SmartLamp:
    brand = "iGuzzini"
    power_watts = 32
```

Рисунок 4.1 – Приклад оголошення класу

На рисунку 4.1 показаний базовий синтаксис оголошення класу в мові *Python*. Опис класу починається зі службового слова *class*. Після ключового слова вказується ім'я класу (*SmartLamp*), яке, відповідно до рекомендацій, зазвичай починається з великої літери.

У тілі класу можуть бути визначені атрибути, які описують властивості майбутніх об'єктів. У наведеному прикладі атрибут *brand* містить назву виробника світильника, а атрибут *power_watts* зберігає значення його номінальної потужності у ватах. Такі атрибути називаються атрибутами класу та є спільними для всіх об'єктів, створених на основі цього класу.

Важливою особливістю *Python* є використання відступів для формування структури програми. Усі атрибути та методи, що належать до класу, повинні розташовуватися всередині його тіла з однаковим відступом відносно заголовка класу.

Після оголошення класу на його основі можна створювати об'єкти. Процес створення об'єкта називається створенням екземпляра класу. Для цього використовується ім'я класу з круглими дужками, після чого створений об'єкт зазвичай присвоюється змінній.

```
# Оголошення класу, що містить лише атрибути
class SmartLamp:
    brand = "iGuzzini"
    power_watts = 32

# Створення об'єкта (екземпляра класу)
lamp = SmartLamp()
```

Рисунок 4.2 – Приклад створення об'єкта

Під час створення об'єкта в оперативній пам'яті виділяється окрема область для зберігання його даних. Кожен об'єкт є самостійним екземпляром класу та може мати власні значення атрибутів, що не залежать від інших об'єктів того самого класу.

Змінна, якій присвоєно створений об'єкт, містить посилання на цей об'єкт і використовується для подальшого доступу до його атрибутів і методів. Після створення об'єкта звернення до його властивостей та виклик методів здійснюються за допомогою оператора крапки «.», який є основним засобом взаємодії з об'єктами в *Python*.

За допомогою оператора крапки можна не лише отримувати значення атрибутів об'єкта та викликати його методи, а й змінювати значення атрибутів під час виконання програми. Для цього після імені об'єкта вказується ім'я атрибута та нове значення, яке необхідно йому присвоїти.

Такий механізм дозволяє змінювати стан об'єкта та налаштовувати його характеристики відповідно до потреб програми.

Завдяки цьому різні об'єкти одного й того самого класу можуть мати власні значення атрибутів незважаючи на те, що були створені на основі спільного шаблону – класу.

```
# 1. Оголошення класу
class SmartLamp:
    brand = "iGuzzini"
    power_watts = 32

# 2. Створення об'єкта
lamp = SmartLamp()

# 3. Читання атрибута та запис його значення в окрему змінну
current_power = lamp.power_watts

# 4. Перезапис атрибута через математичну операцію (збільшуємо потужність на 8 Вт)
lamp.power_watts = lamp.power_watts + 8

# 5. Розрахунок нового значення у зовнішній змінній
boosted_power = lamp.power_watts
```

Рисунок 4.3 – Приклад звернення до властивостей атрибутів через об'єкт

Окрім атрибутів, які використовуються для зберігання даних, класи можуть містити методи. Якщо атрибути визначають стан об'єкта, то методи описують його поведінку, тобто дії, які об'єкт може виконувати.

Методом називають функцію, оголошену всередині класу. На відміну від звичайних функцій, методи безпосередньо пов'язані з об'єктами певного класу та можуть отримувати доступ до їхніх атрибутів. Завдяки цьому за допомогою методів не лише обробляють дані об'єкта, а й змінюють його внутрішній стан.

Оголошення методу в *Python* виконується аналогічно до оголошення звичайної функції за допомогою службового слова *def*.

Під час оголошення методів класу в *Python* першим параметром зазвичай вказується параметр *self*. Він є посиланням на поточний об'єкт класу, для якого було викликано відповідний метод.

За допомогою цього параметра метод отримує доступ до атрибутів та інших методів об'єкта. Параметр *self* не потрібно передавати під час виклику методу. *Python* робить це автоматично, коли метод викликається через об'єкт. Незважаючи на те що ім'я *self* не є зарезервованим словом мови, його використання є загальноприйнятим стандартом програмування в *Python*.

Звернення до атрибутів та методів об'єкта всередині класу здійснюється через конструкцію *self.ім'я_атрибута* або *self.ім'я_методу()*. Це дозволяє однозначно вказати, що доступ виконується саме до даних або методів поточного об'єкта.

На рисунку 4.3 показано приклад використання методів та параметра *self* для доступу до атрибутів об'єкта та їх змінювання всередині методів класу.

У методах *turn_on()* та *boost_power()* звернення до атрибутів здійснюється через конструкції *self.is_active* та *self.power_watts*. Використання параметра *self* дозволяє вказати, що операції виконуються з атрибутами саме того об'єкта, який викликав відповідний метод. Завдяки цьому кожен об'єкт класу може мати власний стан і незалежно змінювати свої характеристики.

Також наведений приклад демонструє можливість передавання параметрів до методів класу. Метод *boost_power()* приймає додатковий параметр *extra_watts*, значення якого передається під час виклику методу. Наприклад, при виконанні виклику *lamp.boost_power(12)* значення 12 передається в параметр *extra_watts*, після чого додається до поточного значення атрибута *power_watts*. Це дозволяє змінювати характеристики об'єкта залежно від вхідних даних без необхідності створення нових методів для кожного окремого випадку.

```

# 1. Оголошення класу з атрибутами та методами
class SmartLamp:
    brand = "iGuzzini"
    power_watts = 32
    is_active = False # Початковий стан: вимкнено

    # Метод для зміни стану (ввімкнення світильника)
    def turn_on(self):
        self.is_active = True

    # Метод для форсування потужності (приймає додатковий аргумент)
    def boost_power(self, extra_watts):
        self.power_watts = self.power_watts + extra_watts

# 2. Створення об'єкта
lamp = SmartLamp()

# 3. Виклик методів для зміни внутрішнього стану об'єкта
lamp.turn_on()          # Змінює прапорець self.is_active на True
lamp.boost_power(12)     # Збільшує self.power_watts з 32 до 44

```

Рисунок 4.4 – Приклад використання методів та параметру *self*

4.3 Конструктори, деструктори

Під час створення об'єктів зазвичай виникає необхідність автоматично виконати певні дії, наприклад, задати початкові значення атрибутів або виконати підготовчі операції перед подальшою роботою об'єкта. Аналогічно після завершення використання об'єкта іноді потрібно звільнити зайняті ресурси або виконати завершальні дії. Для вирішення цих завдань в об'єктно-орієнтованому програмуванні використовуються спеціальні методи, які називаються конструкторами та деструкторами.

Конструктор – це спеціальний метод класу, який автоматично викликається під час створення нового об'єкта. Основним призначенням конструктора є початкова ініціалізація об'єкта, тобто задання початкових значень його атрибутів та підготовка до подальшого використання.

У мові *Python* конструктор реалізується за допомогою спеціального методу `__init__()`. Назва цього методу містить два символи підкреслення –

на початку та в кінці. Після створення об'єкта інтерпретатор *Python* автоматично викликає метод `__init__()`, тому його не потрібно запускати вручну.

Першим параметром конструктора, як і будь-якого іншого методу класу, є параметр *self*, який містить посилання на поточний об'єкт. Крім параметра *self*, конструктор може приймати довільну кількість додаткових параметрів, необхідних для ініціалізації атрибутів об'єкта. У такому випадку під час створення об'єкта відповідні значення передаються через круглі дужки після імені класу.

Використання конструктора дозволяє одразу після створення об'єкта задати його початковий стан, що робить роботу з класами більш зручною та забезпечує коректну ініціалізацію даних.

```
# 1. Оголошення класу з конструктором
class SmartLamp:
    # Конструктор класу – викликається автоматично при створенні об'єкта
    def __init__(self, brand_name, power_value):
        # Ініціалізація унікальних атрибутів для кожного конкретного екземпляра
        self.brand = brand_name
        self.power_watts = power_value
        self.is_active = False

# 2. Створення РІЗНИХ об'єктів з унікальними параметрами через конструктор
lamp_office = SmartLamp("iGuzzini", 32)
lamp_corridor = SmartLamp("Philips", 16)

# 3. Зчитування унікальних даних у змінні
office_brand = lamp_office.brand          # Містить "iGuzzini"
corridor_power = lamp_corridor.power_watts # Містить 16
```

Рисунок 4.5 – Приклад використання конструктора

Деструктор – це спеціальний метод класу, який автоматично викликається під час видалення об'єкта або завершення його життєвого циклу. Основним призначенням деструктора є виконання завершальних дій, таких як звільнення ресурсів, очищення пам'яті або фіксація завершення роботи об'єкта.

У мові *Python* деструктор реалізується за допомогою спеціального методу `__del__()`. Як і інші спеціальні методи, його назва містить два символи підкреслення на початку та в кінці. Деструктор викликається автоматично інтерпретатором, тому його явний виклик у програмі зазвичай не потрібний.

Метод `__del__()` може не приймати обов'язкових параметрів, окрім параметра *self*, який містить посилання на об'єкт, що видаляється. Під час виконання цього методу можна виконувати будь-які дії, пов'язані із завершенням роботи об'єкта.

Потрібно зазначити, що в *Python* використання деструкторів обмежене, оскільки керування пам'яттю здійснюється автоматично за допомогою механізму збирання сміття (*garbage collector*), тому деструктори застосовуються переважно для додаткових дій, а не для прямого управління пам'яттю.

```
# Оголошення класу з конструктором та деструктором
class SystemController:
    def __init__(self, port_number):
        self.port = port_number
        print(f"Контролер: З'єднання на порту {self.port} УСПІШНО ВСТАНОВЛЕНО.")

    # Деструктор класу – викликається автоматично при видаленні об'єкта
    def __del__(self):
        print(f"Деструктор: З'єднання на порту {self.port} БЕЗПЕЧНО РОЗІРВАНО.")

# 1. Створення об'єкта (спрацьовує __init__)
controller = SystemController(8080)

# 2. Явне видалення об'єкта з пам'яті (спрацьовує __del__)
del controller
```

Рисунок 4.6 – Приклад використання деструктора

4.4 Спадкування

Спадкування (наслідування) в ООП дозволяє створювати нові класи на основі вже існуючих. При цьому новий клас (дочірній або похідний) отримує

доступ до атрибутів і методів батьківського (базового) класу та може їх розширювати або перевизначати відповідно до своїх потреб.

```
# Базовий (батьківський) клас
class Luminaire:
    def __init__(self, brand, power):
        self.brand = brand
        self.power_watts = power

    def switch_on(self):
        # Метод, який унаслідують усі нащадки
        self.is_on = True

# Похідний (дочірній) клас спадкує все від Luminaire
class SmartLuminaire(Luminaire):
    # Додаємо новий метод, характерний лише для смарт-світильників
    def set_dimmability(self, percentage):
        self.brightness = percentage

# 1. Створення об'єкта дочірнього класу
smart_lamp = SmartLuminaire("iGuzzini", 32)

# 2. Виклик успадкованого методу від батьківського класу
smart_lamp.switch_on()      # Працює, бо клас унаслідував цей метод

# 3. Виклик власного методу дочірнього класу
smart_lamp.set_dimmability(70)
```

Рисунок 4.7 – Приклад використання спадкування у класі

У мові *Python* спадкування реалізується шляхом зазначення батьківського класу в круглих дужках під час оголошення нового класу. Наприклад, конструкція *class SmartLuminaire (Luminaire)* означає, що клас *SmartLuminaire* є дочірнім щодо класу *Luminaire* і автоматично отримує доступ до його атрибутів та методів.

Завдяки механізму спадкування забезпечується повторне використання коду. Дочірній клас може використовувати вже реалізовані в батьківському класі методи та властивості без необхідності їх повторного опису. Наприклад, об'єкт дочірнього класу автоматично містить атрибути *brand* та *power_watts*, які були визначені в базовому класі, а також може використовувати його методи, зокрема *switch_on*.

Водночас спадкування не обмежується лише копіюванням функціональності. Дочірній клас може розширювати можливості батьківського класу шляхом додавання нових методів або перевизначення існуючих. Наприклад, метод *set_dimmability*, який відповідає за регулювання яскравості світильника, є специфічним для дочірнього класу і недоступний у базовому класі. При спробі виклику цього методу для об'єкта батьківського класу виникне помилка, що підтверджує різницю між рівнями ієрархії класів.

Отже, спадкування дозволяє повторно використовувати вже створений програмний код, який зменшує його дублювання та підвищує гнучкість і масштабованість програмних систем. Дочірній (похідний) клас може:

- використовувати властивості та методи базового класу без їх зміни;
- перевизначати методи базового класу з метою зміни або уточнення їх поведінки;
- додавати нові властивості та методи, які є специфічними лише для цього класу.

У мові *Python* для роботи із спадкуванням використовується вбудована функція *super()*, яка дозволяє звертатися до методів батьківського класу зсередини дочірнього класу. Це особливо важливо у випадках, коли необхідно розширити або доповнити поведінку успадкованого методу, не переписуючи його повністю.

Основне призначення *super()* полягає в тому, щоб забезпечити коректний доступ до реалізації методів базового класу та уникнути дублювання коду при перевизначенні методів у дочірніх класах.

Найчастіше *super()* використовується всередині конструктора *__init__()*, коли дочірній клас повинен ініціалізувати атрибути батьківського класу та додати власні параметри.

```

# Базовий клас
class Luminaire:
    def __init__(self, brand, power):
        self.brand = brand
        self.power_watts = power

# Дочірній клас
class SmartLuminaire(Luminaire):
    def __init__(self, brand, power, protocol):
        # super() викликає конструктор __init__ батьківського класу Luminaire
        super().__init__(brand, power)

        # Додаємо унікальний атрибут дочірнього класу
        self.protocol = protocol # Наприклад: "DALI", "Zigbee"

# Створення об'єкта дочірнього класу
smart_lamp = SmartLuminaire("iGuzzini", 32, "DALI")

# Перевірка: об'єкт має і базові атрибути, і свій власний
device_brand = smart_lamp.brand # Містить "iGuzzini"
device_protocol = smart_lamp.protocol # Містить "DALI"

```

Рисунок 4.8 – Приклад використання функції `super()`

У прикладі на рисунку 4.8 запис `super().__init__(brand, power)` означає, що інтерпретатор знаходить конструктор батьківського класу та виконує його для поточного об'єкта, передаючи відповідні параметри. Таким чином, забезпечується ініціалізація тих атрибутів, які вже були визначені в базовому класі.

Основною перевагою використання `super()` є уникнення дублювання коду. Замість повторного опису однакових рядків ініціалізації в дочірньому класі використовується вже реалізована логіка батьківського класу.

Після виклику `super()` дочірній клас може розширювати функціональність, додаючи власні атрибути та методи, які характерні лише для нього. Наприклад, після ініціалізації базових властивостей можна додати нові змінні, такі як `self.protocol`, що визначають специфічні характеристики дочірнього класу.

Також в ООП є таке поняття як множинне спадкування. Множинне спадкування – це механізм, який дозволяє одному класу наслідувати

властивості та методи одразу від кількох батьківських класів. Це надає можливість об'єднувати функціональність різних класів у межах одного дочірнього класу.

У *Python* множинне спадкування реалізується шляхом зазначення декількох батьківських класів у круглих дужках через кому під час оголошення класу.

```
# Перший базовий клас – відповідає за електрику
class ElectricalDevice:
    def __init__(self, voltage=220):
        self.voltage = voltage

    def check_power(self):
        self.is_safe = True

# Другий базовий клас – відповідає за мережеві функції
class NetworkNode:
    def __init__(self, ip_address="192.168.1.10"):
        self.ip = ip_address

    def connect_to_server(self):
        self.is_connected = True

# Дочірній клас, що наслідує ОДРАЗУ ДВА класи
class SmartController(ElectricalDevice, NetworkNode):
    def __init__(self, voltage, ip_address, model_name):
        # Ініціалізуємо логіку кожного з батьківських класів окремо
        ElectricalDevice.__init__(self, voltage)
        NetworkNode.__init__(self, ip_address)
        self.model = model_name

# Створення об'єкта дочірнього класу
iot_hub = SmartController(230, "192.168.0.50", "Hub-M1")

# Об'єкт має доступ до методів обох батьківських класів
iot_hub.check_power()      # Метод від ElectricalDevice
iot_hub.connect_to_server() # Метод від NetworkNode
```

Рисунок 4.9 – Приклад використання множинного спадкування

Крім множинного спадкування, в об'єктно-орієнтованому програмуванні виокремлюють також інші його різновиди, зокрема багаторівневе спадкування.

Багаторівневе спадкування – це тип наслідування, при якому створюється послідовний ланцюг класів, де кожен наступний клас є похідним від попереднього. У результаті формується ієрархія, в якій кожен рівень успадковує властивості та методи свого безпосереднього батьківського класу, а також усіх класів, що містяться вище в ланцюжку.

```
# Рівень 1: Базовий (корневий) клас
class Device:
    def __init__(self, serial_number):
        self.sn = serial_number

# Рівень 2: Проміжний клас (нащадок Device)
class Luminaire(Device):
    def __init__(self, serial_number, power):
        # Ініціалізуємо базовий пристрій
        super().__init__(serial_number)
        self.power_watts = power

# Рівень 3: Похідний клас (нащадок Luminaire)
class SmartLuminaire(Luminaire):
    def __init__(self, serial_number, power, protocol):
        # Ініціалізуємо проміжний клас Luminaire
        super().__init__(serial_number, power)
        self.protocol = protocol # Наприклад, "DALI"
```

Рисунок 4.10 – Приклад використання багаторівневого спадкування

5 СТРУКТУРИ ДАНИХ

5.1 Списки

Під час програмування зазвичай виникає потреба зберігати та опрацьовувати не окремі значення, а їх сукупність. У багатьох мовах програмування для цього використовуються масиви – структури даних, що складаються з фіксованої кількості елементів, які розміщуються в пам'яті комп'ютера послідовно, а доступ до них здійснюється за допомогою індексів.

У мові *Python* аналогом масивів є списки (*lists*), більш універсальна структура даних. На відміну від класичних масивів, списки не мають фіксованого розміру, тому кількість їх елементів може змінюватися під час виконання програми. Крім того, елементами списку можуть бути об'єкти різних типів даних, що значно розширює можливості їх використання.

Список (list) – це впорядкована, змінювана та гетерогенна структура даних, призначена для зберігання послідовності елементів. Елементами списку можуть бути числа, рядки, логічні значення, інші списки, об'єкти та будь-які інші типи даних. До того ж один список може одночасно містити елементи різних типів.

Ключові характеристики списку:

1. Впорядкованість – елементи зберігаються у певній послідовності, яку *Python* запам'ятовує.
2. Змінюваність – можна змінювати склад і значення елементів після створення.
3. Індксація – кожен елемент має номер (індекс), який починається з позначення 0.
4. Універсальність типів – в одному списку можуть зберігатися елементи різних типів (*int*, *str*, *float*, інші списки тощо).
5. Динамічність – список автоматично змінює розмір під час додавання або видалення елементів.

У *Python* список записується у квадратних дужках (`[]`), а його елементи відокремлюються один від одного комами.

```
# Список із 3 цілих чисел
numbers = [1, 2, 5]

print(numbers)

# Результат: [1, 2, 5]
```

Рисунок 5.1 – Приклад створення списку

Завдяки своїй гнучкості списки дозволяють створювати структури даних різної складності, легко змінювати їх вміст, додавати або видаляти елементи, а також використовувати вкладені списки для побудови багаторівневих колекцій даних.

Крім безпосереднього переліку елементів у квадратних дужках, у *Python* список можна створити на основі інших структур даних або ітерованих об'єктів. Для цього використовується вбудована функція `list()`, яка перетворює передану послідовність на список.

Функція `list()` може приймати різні типи даних, зокрема рядки, кортежі, множини, словники та інші ітеровані об'єкти. У результаті її роботи формується новий список, елементи якого відповідають елементам вихідної структури даних.

```
# 1. Створення списку з текстового рядка
text = "DALI"
chars_list = list(text)

print(chars_list)

# Результат: ['D', 'A', 'L', 'I']
```

Рисунок 5.2 – Створення списку з інших структур даних

У процесі програмування зазвичай виникає потреба об'єднати два або більше списків в один. Така операція називається конкатенацією. У мові *Python* конкатенація списків виконується здебільшого за допомогою перевизначеного оператора додавання (+), який створює новий список, що містить елементи всіх об'єднаних списків.

Під час використання оператора (+) вихідні списки не змінюються. У результаті його роботи формується новий список, елементи якого розташовуються в тому самому порядку, у якому вони були записані в початкових списках.

```
# Спочатку створюємо два окремі списки
emergency_lamps = [4, 8]
office_lamps = [12, 24, 32]

# Об'єднуємо їх за допомогою оператора "+"
all_lamps = emergency_lamps + office_lamps

print(all_lamps)

# Результат: [4, 8, 12, 24, 32]
```

Рисунок 5.3 – Приклад конкатенації списків

У *Python* передбачено можливість багаторазового повторення елементів списку. Для цього використовується перевизначений оператор множення (*), який дозволяє створити новий список, у якому початкова послідовність елементів повторюється задану кількість разів.

Під час виконання операції множення вихідний список не змінюється. Результатом є новий список, що містить елементи початкового списку, повторені відповідно до вказаного числа.

```
# Базовий шаблон рівнів яскравості для однієї зони
zone_pattern = [50, 100]

# Повторюємо цей шаблон 3 рази за допомогою оператора "*"
full_schedule = zone_pattern * 3

print(full_schedule)

# Результат: [50, 100, 50, 100, 50, 100]
```

Рисунок 5.4 – Приклад багаторазового повторення елементів списку

Однією з основних переваг списків є можливість швидкого доступу до окремих елементів за допомогою індексів. Кожний елемент списку має власний порядковий номер – індекс, який визначає його положення в послідовності.

У *Python* нумерація елементів починається з нуля, тобто перший елемент має індекс 0, другий – 1, третій – 2 і т. д. Для звернення до елемента необхідно після імені списку в квадратних дужках вказати його індекс.

```
# Створення списку брендів світильників
brands = ["iGuzzini", "Philips", "Eaton"]

# Звернення до першого елемента (індекс 0)
first_item = brands[0]
print(first_item)

# Результат: iGuzzini
```

Рисунок 5.5 – Доступ до елементів списку в *Python* за допомогою додатної індексації

Крім додатних індексів, *Python* підтримує від’ємну індексацію, яка дозволяє звертатися до елементів починаючи з кінця списку. При цьому останній елемент має індекс -1, передостанній -2, третій з кінця -3 і т. д.

```
# Створення списку брендів обладнання
brands = ["iGuzzini", "Philips", "Eaton", "Osram"]

# Звернення до останнього елемента списку
last_item = brands[-1]
print(last_item)

# Результат: Osram
```

Рисунок 5.6 – Доступ до елементів списку в *Python* за допомогою від’ємної індексації

За допомогою індексів можна не лише отримувати значення елементів, а й змінювати їх. Для цього достатньо звернутися до потрібного елемента за індексом та присвоїти йому нове значення.

```
# Створення початкового списку брендів
brands = ["iGuzzini", "Philips", "Eaton"]

# Заміна другого елемента (індекс 1) на новий бренд
brands[1] = "Osram"

print(brands)

# Результат: ['iGuzzini', 'Osram', 'Eaton']
```

Рисунок 5.7 – Приклад перезапису та оновлення окремого елемента всередині списку за його індексом

Якщо звернутися до індексу, який виходить за межі списку, інтерпретатор *Python* згенерує помилку *IndexError*, оскільки елемент із таким індексом не існує.

Під час роботи зі списками зазвичай виникає потреба отримати не окремий елемент, а певну частину списку. Для цього в *Python* використовуються зрізи (*slices*), які дозволяють створити новий список, що містить обраний діапазон елементів початкового списку.

Зріз формується за допомогою оператора «:» (двокрапка) у квадратних дужках після імені списку. Загальний формат запису має такий вигляд:

список[початок:кінець:крок],

де початок – індекс першого елемента, який необхідно включити до зрізу; кінець – індекс елемента, до якого виконується вибірка (цей елемент не входить до зрізу); крок – інтервал між елементами, що обираються. Якщо крок не вказано, за замовчуванням використовується значення 1.

Наприклад, наведений нижче код створює зріз, що містить елементи з індексами від 1-го до 3-х включно:

```
numbers = [10, 20, 30, 40, 50, 60]
print(numbers[1:4]) # [20, 30, 40]
```

Якщо початковий індекс не вказано, зріз починається з першого елемента списку:

```
print(numbers[:3]) # [10, 20, 30]
```

Якщо не вказаний кінцевий індекс, зріз міститиме всі елементи від зазначеної позиції до кінця списку:

```
print(numbers[2:]) # [30, 40, 50, 60]
```

Якщо не зазначити ні початковий, ні кінцевий індекси, буде створено копію всього списку:

```
copy_numbers = numbers[:]
```

Параметр «крок» дозволяє обирати елементи через певний інтервал:

```
print(numbers[::2]) # [10, 30, 50]
```

Також крок може бути від’ємним. У такому випадку перегляд списку виконується у зворотному напрямі:

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

Окрім операцій індексації, зрізів та об’єднання, мова *Python* надає широкий набір вбудованих методів, які значно спрощують роботу зі списками. З їх допомогою можна додавати нові елементи, вставляти їх у довільне місце списку, видаляти, змінювати порядок розташування, сортувати, виконувати пошук та здійснювати інші операції з елементами тощо.

Методи (табл. 5.1) є функціями, що належать об’єктам типу *list*, тому вони викликаються за допомогою оператора крапки (.) після імені списку.

Кожний метод призначений для виконання певної операції зі списком, що дозволяє ефективно керувати його вмістом без необхідності реалізації відповідних алгоритмів вручну.

Таблиця 5.1 – Методи для роботи зі списками

Метод	Опис
<i>append()</i>	Додає елемент у кінець списку
<i>extend()</i>	Додає елементи зі списку в кінець іншого списку
<i>insert()</i>	Вставляє елемент за вказаним індексом
<i>remove()</i>	Видаляє елемент за вказаним значенням
<i>pop()</i>	Повертає та видаляє елемент, присутній за вказаним індексом
<i>clear()</i>	Видаляє всі елементи зі списку
<i>index()</i>	Повертає індекс вказаного елемента
<i>count()</i>	Повертає кількість вказаних елементів у списку
<i>sort()</i>	Сортує список у порядку зростання / спадання
<i>reverse()</i>	Повертає список у зворотному порядку («розвертає» послідовність)
<i>copy()</i>	Повертає поверхневу копію списку

Однією з найпоширеніших операцій під час роботи зі списками є *ітерація*, тобто послідовний перегляд усіх елементів списку. У *Python* для цього зазвичай використовується цикл *for*, який автоматично перебирає кожний елемент колекції без необхідності керувати індексами.

Для перевірки, чи містить список певний елемент, використовується оператор належності *in*. Якщо зазначений елемент присутній у списку, результатом буде логічне значення *True*, якщо ні – *False*.

Для визначення кількості елементів у списку використовується вбудована функція `len()`. Вона повертає ціле число, що відповідає кількості елементів у списку.

```
# Створення списку з номерами робочих каналів керування
channels = [10, 12, 15]

# Перебір елементів списку за допомогою циклу for
for ch in channels:
    print(ch)

# Результат:
# 10
# 12
# 15
```

Рисунок 5.8 – Приклад застосування ітерації у списках

```
# Створення списку з адресами шлюзів керування
gateways = ["192.168.1.1", "192.168.1.5", "192.168.1.10", "192.168.1.20"]

# Визначення кількості елементів у списку
total_count = len(gateways)

print(total_count)

# Результат: 4
```

Рисунок 5.9 – Приклад визначення кількості елементів у списку

5.2 Кортежі

Окрім списків у *Python* існує ще один тип даних для зберігання послідовностей елементів – кортеж (*tuple*). За своєю структурою кортеж дуже схожий на список, проте має одну принципову відмінність: після створення його вміст не можна змінити. Саме тому кортеж зазвичай називають незмінюваним списком.

Кортеж (tuple) – це впорядкована, незмінювана структура даних, призначена для зберігання послідовності елементів. Як і списки, кортежі можуть містити об’єкти різних типів даних, зокрема числа, рядки, логічні значення, інші кортежі та інші об’єкти *Python*.

У *Python* кортеж записується у круглих дужках `()`, а його елементи відокремлюються комами.

```
# Створення кортежу з координатами світлової точки (X, Y)
point_coordinates = (15.5, 24.0)

print(point_coordinates)

# Результат: (15.5, 24.0)
```

Рисунок 5.10 – Приклад створення кортежу

Хоча кортежі та списки мають схожу структуру, між ними існують суттєві відмінності, які визначають доцільність їх використання в різних ситуаціях.

Головною перевагою кортежів є їх незмінність. Після створення кортежу неможливо додавати, видаляти або змінювати його елементи. Завдяки цьому кортежі доцільно використовувати для зберігання даних, які не повинні змінюватися під час виконання програми. Це дозволяє уникнути випадкової модифікації даних і підвищує надійність програмного коду.

На відміну від списків, які зазвичай використовуються для зберігання однотипних даних, кортежі застосовуються здебільшого для об'єднання кількох взаємопов'язаних значень різних типів. Наприклад, за допомогою кортежу можна зберігати координати точки, інформацію про студента або інші набори фіксованих характеристик.

Оскільки кортежі є незмінюваними структурами даних, операції їх обробки, зокрема ітерація, можуть виконуватися дещо швидше, ніж для списків. Крім того, кортежі, як правило, займають менше оперативної пам'яті, що робить їх більш ефективними для зберігання незмінних даних.

Ще однією важливою перевагою є можливість використання кортежів як ключів словників за умови, що всі їх елементи також є незмінюваними. Списки не можуть використовуватися як ключі словників, оскільки належать до змінюваних типів даних.

Незважаючи на незмінюваність, кортежі підтримують більшість операцій, характерних для послідовностей. До них можна застосовувати індексацію, зрізи, операції конкатенації (+) та повторення (*), ітерації, перевірку на існування елемента в кортежі (*in*), а також використовувати вбудовані функції *len()*, *min()*, *max()*, методи *count()* і *index()*. При цьому всі операції, які змінюють послідовність, для кортежів недоступні.

Окрім безпосереднього переліку елементів у круглих дужках, у *Python* кортеж можна створити на основі інших структур даних або ітерованих об'єктів. Для цього використовується вбудована функція *tuple()*, яка перетворює передану послідовність на кортеж.

5.3 Словники

Під час роботи зі списками доступ до елементів здійснюється за допомогою числових індексів. Такий підхід є зручним, коли важливим є порядок розташування елементів. Проте в багатьох завданнях значно зручніше звертатися до даних не за їхнім порядковим номером, а за певною назвою або ідентифікатором. Саме для цього в *Python* використовуються словники.

Словник (*dictionary*, *dict*) – це змінювана структура даних, призначена для зберігання інформації у вигляді пар «ключ – значення». Кожне значення пов'язане з унікальним ключем, за допомогою якого до нього можна швидко отримати доступ.

У *Python* словники записуються у фігурних дужках {}, а кожен елемент подається у форматі *ключ : значення*. Пари «ключ – значення» відокремлюються одна від одної комами.

Ключі словника повинні бути унікальними, тобто в межах одного словника не може існувати двох однакових ключів. Крім того, як ключі можуть використовуватися лише об'єкти незмінюваних типів даних,

наприклад, цілі числа, дійсні числа, рядки, логічні значення або кортежі. Значення словника можуть мати будь-який тип даних, зокрема списки, кортежі, інші словники та будь-які інші об'єкти *Python*.

```
# Створення словника, де:
# Ключі (ліворуч від двокрапки): "brand", "power_watts"
# Значення (праворуч від двокрапки): "iGuzzini", 32

lamp_info = {
    "brand": "iGuzzini",      # Ключ: "brand",      Значення: "iGuzzini"
    "power_watts": 32        # Ключ: "power_watts", Значення: 32
}

print(lamp_info)

# Результат: {'brand': 'iGuzzini', 'power_watts': 32}
```

Рисунок 5.11 – Приклад створення словника

Окрім створення словника за допомогою фігурних дужок {}, у *Python* існує ще один спосіб – використання вбудованої функції *dict()*. Вона дозволяє створити словник, передавши до нього іменовані аргументи, де назви аргументів автоматично стають ключами словника, а їх значення – відповідними значеннями.

На відміну від списків, доступ до елементів словника здійснюється не за індексом, а за ключем. Кожне значення у словнику пов'язане з унікальним ключем, тому для отримання необхідного значення достатньо вказати ключ у квадратних дужках після імені словника.

```
# Створення словника
lamp_info = {"brand": "iGuzzini", "power_watts": 32}

# Звернення до елемента за ключем "brand"
current_brand = lamp_info["brand"]
print(current_brand)
# Результат: iGuzzini

# Звернення до елемента за ключем "power_watts"
current_power = lamp_info["power_watts"]
print(current_power)
# Result: 32
```

Рисунок 5.12 – Отримання даних зі словника за допомогою ключа

Якщо під час звернення вказати ключ, якого немає у словнику, *Python* згенерує виняток *KeyError*.

Як і для інших структур даних у *Python*, до словників можна застосувати низку універсальних операцій. Зокрема, для перевірки наявності елемента використовується оператор *in*, для послідовного опрацювання елементів – цикл *for*, а для визначення кількості елементів словника – вбудована функція *len()*.

Під час використання оператора *in* перевіряється наявність ключа у словнику. Якщо зазначений ключ існує, результатом буде *True*, якщо ні – *False*. Для перебору елементів словника зазвичай використовується цикл *for*. За замовчуванням цикл перебирає ключі словника, після чого за кожним ключем можна отримати відповідне значення. Для визначення кількості елементів словника використовується функція *len()*, яка повертає кількість пар «ключ – значення», що містяться у словнику.

Потрібно зазначити, що словники в *Python* є змінюваними структурами даних, тому після їх створення можна додавати нові елементи або змінювати існуючі. Додавання нового елемента до словника здійснюється шляхом присвоєння значення новому ключу. Якщо вказаний ключ не існує у словнику, то буде створено нову пару «ключ – значення». Якщо ж ключ уже існує у словнику, то відбудеться перезапис значення, тобто попереднє значення буде замінено новим.

```
# Створення початкового словника з базовими параметрами
lamp_info = {"brand": "iGuzzini", "power_watts": 32}

# Додавання нового елемента: ключ "protocol", значення "DALI"
lamp_info["protocol"] = "DALI"

print(lamp_info)

# Результат: {'brand': 'iGuzzini', 'power_watts': 32, 'protocol': 'DALI'}
```

Рисунок 5.13 – Додавання нового елемента до словника

Також можна не лише додавати елементи до словника, але й видаляти їх. Для видалення елементів зі словника використовується кілька способів, найпоширенішим із яких є оператор *del*. Оператор *del* дозволяє видалити елемент словника за його ключем.

```
# Створення початкового словника з параметрами пристрою
lamp_info = {"brand": "iGuzzini", "power_watts": 32, "protocol": "DALI"}

# Видалення елемента за ключем "protocol" за допомогою оператора del
del lamp_info["protocol"]

print(lamp_info)

# Результат: {'brand': 'iGuzzini', 'power_watts': 32}
```

Рисунок 5.14 – Видалення елементів зі словника за допомогою оператора *del*

Під час роботи зі словниками в *Python* використовується багато різних вбудованих методів (табл. 5.1), які дозволяють зручно виконувати основні операції з даними. Завдяки цим методам можна отримувати доступ до ключів і значень, додавати нові елементи, видаляти існуючі, а також змінювати вміст словника без необхідності написання додаткових алгоритмів тощо.

Таблиця 5.2 – Методи для роботи зі словниками

Метод	Призначення
1	2
<i>dict.keys()</i>	Повертає всі ключі словника
<i>dict.values()</i>	Повертає всі значення словника
<i>dict.items()</i>	Повертає пари «ключ – значення»
<i>dict.get(key[, default])</i>	Повертає значення за ключем, якщо немає – <i>default</i>
<i>dict.pop(key[, default])</i>	Видаляє ключ і повертає значення

Продовження таблиці 5.2

1	2
<i>dict.popitem()</i>	Видаляє та повертає останню пару
<i>dict.update(other_dict)</i>	Додає або оновлює пари словника
<i>dict.clear()</i>	Очищає словник
<i>dict.copy()</i>	Повертає копію словника
<i>dict.setdefault(key, default)</i>	Повертає значення ключа, якщо немає додає <i>default</i>
<i>zip()</i>	Поеднує дві або більше ітерабельні послідовності у пари (можна створювати словники)

5.4 Множини

Множина (*set*) – це структура даних у *Python*, яка становить неупорядкований набір унікальних елементів. За своєю логікою множини дуже подібні до математичних множин і використовуються тоді, коли важлива лише наявність елемента, а не його порядок або кількість повторень.

Множина може містити елементи різних типів даних, проте всі елементи повинні бути незмінюваними. Тобто в множину можуть входити числа, рядки, кортежі, але не можуть входити списки або інші множини.

Це обмеження пов'язане з тим, що множини реалізуються на основі хешування і елементи повинні мати стабільне значення для обчислення хеш-значення.

У *Python* множина є змінюваною структурою даних, тому після створення її можна змінювати: додавати нові елементи або видаляти існуючі.

Множини записуються у фігурних дужках {}, а елементи розділяються комами.

Основні властивості множин:

- елементи не повторюються (дублікати автоматично видаляються);
- порядок елементів не зберігається;
- доступ за індексом відсутній;
- перевірка належності елемента виконується дуже швидко.

```
# Створення звичайної множини брендів без повторів  
brands = {"iGuzzini", "Philips", "Eaton"}  
  
print(brands)  
  
# Результат: {'Eaton', 'Philips', 'iGuzzini'}
```

Рисунок 5.15 – Приклад створення множини

Окрім створення множини за допомогою фігурних дужок {}, у *Python* існує ще один спосіб – використання вбудованої функції *set()*. Вона дозволяє створити множину на основі будь-якої ітерованої структури даних, наприклад списку або рядка.

У разі використання функції *set()* всі повторювані елементи автоматично видаляються, оскільки множина може містити лише унікальні значення.

Як і для інших структур даних у *Python*, для множин також можна використовувати стандартні інструменти роботи з даними, зокрема цикл *for*, функцію *len()* та оператор *in*.

За допомогою циклу *for* виконується перебір усіх елементів множини. При цьому порядок елементів не гарантується, оскільки множина є неупорядкованою структурою даних.

Функція *len()* використовується для визначення кількості елементів у множині та повертає число унікальних значень.

Оператор *in* застосовується для перевірки наявності елемента в множині. Це одна з найефективніших операцій, оскільки множини оптимізовані для швидкої перевірки належності.

У *Python* множини підтримують виконання основних математичних операцій, подібних до операцій із множинами в математиці. Ці операції дозволяють ефективно працювати з наборами даних, порівнювати їх та отримувати нові множини на основі вже існуючих.

Об'єднання множин

Операція об'єднання дозволяє отримати нову множину, яка містить усі унікальні елементи двох множин. Для виконання операції об'єднання множин використовується оператор `|` або метод *union()*.

```
# Перша множина унікальних брендів
zone_a = {"iGuzzini", "Philips"}

# Друга множина унікальних брендів
zone_b = {"Eaton", "Osram"}

# Об'єднання двох множин за допомогою оператора "|"
all_brands = zone_a | zone_b

print(all_brands)

# Результат: {'Eaton', 'Philips', 'Osram', 'iGuzzini'}
```

Рисунок 5.16 – Приклад об'єднання множин

Перетин множин

Операція перетину повертає лише ті елементи, які присутні одночасно в обох множинах. У *Python* для виконання операції перетину множин використовується оператор `&` або метод *intersection()*.

```
# Перша множина брендів обладнання
zone_a = {"iGuzzini", "Philips", "Eaton"}

# Друга множина брендів обладнання
zone_b = {"Philips", "Osram", "Eaton"}

# Пошук перетину двох множин за допомогою оператора "&"
common_brands = zone_a & zone_b

print(common_brands)

# Результат: {'Eaton', 'Philips'}
```

Рисунок 5.17 – Приклад перетину множин

Різниця множин

Операція різниці повертає елементи, які є в першій множині, але відсутні в другій. У *Python* для виконання операції різниці між двома множинами використовується оператор або метод *difference()*.

```
# Перша множина брендів обладнання
zone_a = {"iGuzzini", "Philips", "Eaton"}

# Друга множина брендів обладнання
zone_b = {"Philips", "Osram", "Eaton"}

# Пошук різниці множин (елементи з zone_a, яких немає в zone_b)
unique_to_a = zone_a - zone_b

print(unique_to_a)

# Результат: {'iGuzzini'}
```

Рисунок 5.18 – Приклад різниці множин

Симетрична різниця

Симетрична різниця повертає елементи, які належать лише одній із множин (без спільних елементів). У *Python* для виконання операції симетричної різниці між двома множинами використовується оператор $\wedge$ або метод *symmetric_difference()*.

```

# Перша множина брендів обладнання
zone_a = {"iGuzzini", "Philips", "Eaton"}

# Друга множина брендів обладнання
zone_b = {"Philips", "Osram", "Eaton"}

# Пошук симетричної різниці за допомогою оператора "^"
exclusive_brands = zone_a ^ zone_b

print(exclusive_brands)

# Результат: {'Osram', 'iGuzzini'}

```

Рисунок 5.19 – Приклад симетричної різниці множин

У таблиці 5.3 наведено основні методи роботи з множинами в *Python*. Вони охоплюють широкий спектр операцій і дозволяють виконувати як базові дії (додавання та видалення елементів), так і більш складні операції з множинами, зокрема об'єднання, перетин, різницю та симетричну різницю.

Таблиця 5.3 – Методи для роботи з множинами

Метод	Призначення
1	2
<i>add()</i>	Додає елемент до множини
<i>all()</i>	Повертає <i>True</i> , якщо всі елементи множини дорівнюють <i>True</i> (або якщо множина порожня)
<i>any()</i>	Повертає <i>True</i> , якщо хоч один з елементів множини дорівнює <i>True</i> . Якщо множина порожня, повертається <i>False</i>
<i>clear()</i>	Видаляє всі елементи з множини
<i>copy()</i>	Повертає копію множини

Продовження таблиці 5.3

1	2
<i>difference()</i>	Повертає різницю двох або більше множин у вигляді нової множини
<i>difference_update()</i>	Видаляє всі елементи іншої множини з поточної множини
<i>discard()</i>	Видаляє елемент із множини, якщо він є її частиною (нічого не робить, якщо елемент не міститься у множині)
<i>enumerate()</i>	Повертає об'єкт переліку, який містить індекс і значення для всіх елементів множини у вигляді пари
<i>intersection()</i>	Повертає перетин двох множин у вигляді нової множини
<i>intersection_update()</i>	Оновлює множину шляхом перетину себе та іншої множини
<i>isdisjoint()</i>	Повертає <i>True</i> , якщо дві множини не мають перетину
<i>issubset()</i>	Повертає <i>True</i> , якщо інша множина містить поточну множину
<i>issuperset()</i>	Повертає <i>True</i> , якщо поточна множина містить іншу множину
<i>len()</i>	Повертає довжину (кількість елементів) у множині
<i>max()</i>	Повертає найбільший елемент у множині
<i>min()</i>	Повертає найменший елемент у множині
<i>pop()</i>	Видаляє та повертає довільний елемент множини, викликає помилку <i>KeyError</i> , якщо множина порожня
<i>remove()</i>	Видаляє елемент із множини. Якщо елемент не є членом множини, виникає помилка <i>KeyError</i>

Продовження таблиці 5.3

1	2
<i>sorted()</i>	Повертає новий відсортований список з елементів множини (але не виконує сортування самої множини)
<i>sum()</i>	Повертає суму всіх елементів множини
<i>symmetric_difference()</i>	Повертає симетричну різницю двох множин у вигляді нової множини
<i>symmetric_difference_update()</i>	Оновлює множину з симетричною різницею себе та іншої
<i>union()</i>	Повертає об'єднання множин у новій множині
<i>update()</i>	Оновлює множину з об'єднанням себе та інших

6 МОДУЛІ ТА БІБЛІОТЕКИ

6.1 Загальна інформація про модулі та бібліотеки

Під час створення програм не завжди доцільно реалізовувати всі функції самостійно. *Python* містить багато готових програмних компонентів, які значно спрощують розроблення програм. Такі компоненти об'єднуються у модулі та бібліотеки, що дозволяють повторно використовувати вже створений код, підвищувати продуктивність розробки та зменшувати кількість помилок.

Модуль – це окремий файл із програмним кодом, який містить функції, класи, змінні та інші програмні конструкції. Модулі використовуються для логічного поділу програми на окремі частини та повторного використання коду.

Бібліотека – це сукупність взаємопов'язаних модулів, призначених для вирішення певного класу завдань. *Python* має потужну стандартну бібліотеку, яка встановлюється разом з інтерпретатором та містить модулі для математичних обчислень, роботи з файлами, датою і часом, випадковими числами, операційною системою та багатьма іншими завданнями.

Крім стандартної бібліотеки *Python*, існує багато сторонніх бібліотек, розроблених спільнотою програмістів для вирішення найрізноманітніших завдань. Такі бібліотеки не входять до стандартного набору *Python*, тому перед використанням їх необхідно окремо встановити.

Сторонні (зовнішні) бібліотеки значно розширюють можливості мови програмування та дозволяють виконувати складні обчислення, працювати з великими обсягами даних, створювати графічні інтерфейси, вебзастосунки, системи штучного інтелекту, комп'ютерного зору та багатьох інших програмних продуктів.

Для використання можливостей модулів стандартної бібліотеки його необхідно підімкнути до програми. У *Python* для цього використовується оператор *import*, який повідомляє інтерпретатору, що в програмі будуть використовуватися певні модулі стандартної бібліотеки:

import ім'я_модуля

```
import math
```

Рисунок 6.1 – Приклад підімкнення модуля стандартної бібліотеки

Після підімкнення всі функції, константи та інші об'єкти модуля стають доступними через оператор крапки (*.*), який дозволяє звернутися до потрібного елемента модуля.

Наприклад, для обчислення квадратного кореня з числа використовується функція *sqrt()* модуля *math*.

```
import math

# Обчислення квадратного кореня з числа 25
result = math.sqrt(25)

print(result)
# Результат: 5.0
```

Рисунок 6.2 – Приклад використання модуля стандартної бібліотеки

У цьому прикладі спочатку до програми підмикається модуль *math*, після чого за допомогою запису *math.sqrt(25)* викликається функція *sqrt()*, яка обчислює квадратний корінь із числа 25.

Якщо у програмі необхідно використати лише окремі функції, класи або змінні модуля, доцільно імпортувати лише потрібні об'єкти. Для цього використовується оператор *from ... import*:

from ім'я_модуля import ім'я_об'єкта

Наприклад, якщо необхідно використати лише функцію `sqrt()` з модуля `math`, можна виконати її безпосередній імпорт.

```
from math import sqrt

# Обчислення квадратного кореня (виклик функції напряму)
result = sqrt(25)

print(result)
# Результат: 5.0
```

Рисунок 6.3 – Приклад використання конкретного інструмента з модуля

У цьому випадку після імпорту звертатися до функції через ім'я модуля вже не потрібно. Функцію можна викликати безпосередньо за її назвою.

За допомогою одного оператора можна імпортувати відразу кілька об'єктів модуля, перелічивши їх через кому.

Імпорт окремих об'єктів дозволяє зробити код більш компактним, однак його потрібно використовувати обережно. Якщо в програмі присутні функції або змінні з однаковими іменами, це може призвести до конфлікту назв. Тому такий спосіб імпорту доцільно застосовувати тоді, коли використовується небагато об'єктів із модуля, а їхні назви однозначні.

У *Python* існує можливість підікнути модулі або окремі об'єкти з використанням псевдоніма. Для цього застосовується ключове слово *as*, яке дозволяє задати нове ім'я для модуля або функції. Такий підхід робить код більш компактним і зручним для читання, особливо якщо назва модуля довга або часто використовується в програмі.

Загальний синтаксис перейменування модуля має такий вигляд:

import ім'я_модуля as псевдонім

За допомогою ключового слова *as* можна також перейменувати окремі функції або інші об'єкти модуля.

Загальний синтаксис:

from ім'я_модуля import ім'я_об'єкта as псевдонім

Використання псевдонімів є особливо корисним тоді, коли модуль має довгу назву або коли необхідно уникнути конфлікту імен між об'єктами різних модулів.

Окрім модулів стандартної бібліотеки *Python*, розробники можуть використовувати сторонні (зовнішні) бібліотеки. Оскільки сторонні бібліотеки не входять до складу стандартної бібліотеки *Python*, перед їх використанням необхідно виконати встановлення. Для цього використовується менеджер пакетів *pip*, який автоматично завантажує необхідну бібліотеку та встановлює її у середовище *Python*.

Загальний синтаксис встановлення бібліотеки має такий вигляд:

pip install назва_бібліотеки

Після успішного встановлення бібліотеку можна підімкнути до програми за допомогою оператора *import*.

Якщо бібліотека вже встановлена, повторне виконання команди *pip install* не приведе до її повторного завантаження. Менеджер пакетів автоматично перевірить наявність бібліотеки та повідомить користувача, що вона вже встановлена.

Варто зазначити, що команда *pip install* виконується у командному рядку або терміналі операційної системи, а не безпосередньо у програмному коді *Python*.

```
import numpy

# Створення масиву за допомогою повної назви бібліотеки
my_array = numpy.array([1, 2, 3, 4, 5])

print(my_array)
# Результат: [1 2 3 4 5]
```

Рисунок 6.4 – Приклад використання бібліотеки *Numpy*

Для імпорту також можна використовувати конструкції *from ... import* та *as*, які дозволяють імпортувати окремі об'єкти або задавати їм псевдоніми. Звернення до функцій, класів та інших об'єктів бібліотеки, як правило,

здійснюється за допомогою оператора крапки (.). Таким чином, принцип роботи зі сторонніми бібліотеками практично не відрізняється від роботи зі стандартними модулями *Python*. Відмінність полягає лише в тому, що сторонні бібліотеки необхідно попередньо встановити за допомогою менеджера пакетів *pip*, після чого вони використовуються так само, як і модулі стандартної бібліотеки.

6.2 Модулі стандартної бібліотеки

Розглянемо найбільш поширені модулі стандартної бібліотеки, які зазвичай використовуються під час розроблення програм мовою *Python*.

Модуль *math* є одним із найбільш використовуваних модулів стандартної бібліотеки *Python*. Він містить широкий набір математичних функцій і констант, які дозволяють виконувати різноманітні обчислення без необхідності самостійної реалізації відповідних алгоритмів. Модуль *math* містить функції для обчислення квадратного та кубічного коренів, піднесення до степеня, округлення чисел, обчислення тригонометричних функцій, логарифмів, факторіалів та багатьох інших математичних операцій. Крім цього, модуль містить математичні константи, зокрема число π (*pi*) та число Ейлера (*e*).

```
import math

# 1. Квадратний корінь (sqrt)
square_root = math.sqrt(49)
print(square_root) # Результат: 7.0

# 2. Піднесення до степеня (pow)
# Перший аргумент – число, другий – степінь (2 в кубі)
power_result = math.pow(2, 3)
print(power_result) # Результат: 8.0

# 3. Тригонометрія (косинус кута 60 градусів)
# Спочатку переводимо 60 градусів у радіани
angle_in_radians = math.radians(60)
cosine_result = math.cos(angle_in_radians)
print(cosine_result) # Результат: 0.5000000000000001 (через особливості float)
```

Рисунок 6.5 – Приклад використання модуля *math*

Найбільш поширені функції та константи модуля *math* наведені в таблиці 6.1.

Таблиця 6.1 – Найбільш поширені функції та константи модуля *math*

Функція (метод)	Призначення
<i>sqrt(x)</i>	Обчислення квадратного кореня
<i>pow(x, y)</i>	Піднесення числа до степеня
<i>ceil(x)</i>	Округлення до більшого цілого
<i>floor(x)</i>	Округлення до меншого цілого
<i>fabs(x)</i>	Модуль числа
<i>factorial(n)</i>	Обчислення факторіала
<i>sin(x)</i>	Синус кута (у радіанах)
<i>cos(x)</i>	Косинус кута (у радіанах)
<i>tan(x)</i>	Тангенс кута (у радіанах)
<i>degrees(x)</i>	Перетворення радіанів у градуси
<i>radians(x)</i>	Перетворення градусів у радіани
<i>log(x)</i>	Натуральний логарифм
<i>log10(x)</i>	Десятковий логарифм
<i>exp(x)</i>	Обчислення (e^x)
<i>pi</i>	Число π
<i>e</i>	Число Ейлера

Модуль *random* входить до складу стандартної бібліотеки *Python* і призначений для генерування псевдовипадкових чисел, а також випадкового вибору елементів із послідовностей. Його функції широко використовуються під час створення ігор, моделювання випадкових процесів, тестування програм, статистичних досліджень та багатьох інших завдань.

Модуль *random* містить функції для генерування випадкових цілих і дійсних чисел, випадкового вибору елементів із послідовностей, перемішування елементів списку, а також формування випадкових вибірок. Варто зазначити, що числа, згенеровані модулем *random*, є псевдовипадковими, тобто вони створюються за допомогою спеціальних алгоритмів і лише імітують випадковість.

```

import random

# 1. Випадкове цілочисельне значення (randint)
# Генерує ціле число у вказаному діапазоні включно (від 1 до 10)
random_int = random.randint(1, 10)
print(random_int) # Можливий результат: 7

# 2. Випадкове число з плаваючою крапкою (random)
# Генерує дробове число від 0.0 до 1.0 (не включаючи 1.0)
random_float = random.random()
print(random_float) # Можливий результат: 0.4215748931

# 3. Випадковий вибір зі списку (choice)
# Обирає один довільний елемент із наявної послідовності
brands = ["iGuzzini", "Philips", "Eaton", "Osram"]
selected_brand = random.choice(brands)
print(selected_brand) # Можливий результат: Philips

# 4. Перемішування списку на місці (shuffle)
# Хаотично змінює порядок елементів у самому списку
random.shuffle(brands)
print(brands) # Можливий результат: ['Eaton', 'Osram', 'iGuzzini', 'Philips']

```

Рисунок 6.6 – Приклад використання модуля *random*

Основні функції модуля *random* наведені в таблиці 6.2.

Таблиця 6.2 – Основні функції модуля *random*

Функція (метод)	Призначення
<i>random()</i>	Генерує випадкове дійсне число в діапазоні від 0.0 до 1.0
<i>randint(a, b)</i>	Генерує випадкове ціле число в діапазоні від a до b включно
<i>randrange(start, stop[, step])</i>	Генерує випадкове число із заданого діапазону
<i>uniform(a, b)</i>	Генерує випадкове дійсне число в діапазоні від a до b
<i>choice(seq)</i>	Повертає випадковий елемент з послідовності
<i>choices(seq, k=n)</i>	Повертає список з n випадково обраних елементів (із можливими повтореннями)
<i>sample(seq, k)</i>	Повертає випадкову вибірку з k унікальних елементів
<i>shuffle(seq)</i>	Випадково переміщує елементи списку
<i>seed(x)</i>	Встановлює початкове значення генератора псевдовипадкових чисел

Модуль *datetime* входить до складу стандартної бібліотеки *Python* і призначений для роботи з датою та часом. Він дозволяє створювати об'єкти, що становлять дату, час або їх поєднання, виконувати арифметичні операції з ними, а також формувати їх для зручного відображення.

Модуль *datetime* широко використовується під час розроблення програм, пов'язаних із веденням журналів подій, плануванням завдань, обробкою часових міток, визначенням тривалості подій та іншими завданнями, у яких необхідно працювати з календарними датами та часом.

```
import datetime

# 1. Отримання поточної дати та часу (now)
current_now = datetime.datetime.now()
print(current_now) # Результат: 2026-06-28 14:50:12.451283

# 2. Створення конкретної власної дати (рік, місяць, день)
custom_date = datetime.date(2026, 9, 1)
print(custom_date) # Результат: 2026-09-01

# 3. Форматування дати у звичний текстовий рядок (strftime)
# %d – день, %m – місяць, %Y – повний рік
formatted_date = current_now.strftime("%d.%m.%Y")
print(formatted_date) # Результат: 28.06.2026

# 4. Робота з інтервалами часу (timedelta)
# Додаємо 7 днів до поточної дати
next_week = current_now + datetime.timedelta(days=7)
print(next_week.strftime("%d.%m.%Y")) # Результат: 05-07-2026 (через тиждень)
```

Рисунок 6.7 – Приклад використання модуля *datetime*

Найбільш популярні методи модуля наведені в таблиці 6.3.

Таблиця 6.3 – Популярні методи модуля *datetime*

Функція (метод)	Призначення
1	2
<i>datetime.datetime.now()</i>	Повертає поточні дату та час
<i>datetime.date.today()</i>	Повертає поточну дату
<i>datetime.datetime(year, month, day, hour, minute, second)</i>	Створює об'єкт дати та часу
<i>datetime.date(year, month, day)</i>	Створює об'єкт дати

Продовження таблиці 6.3

1	2
<code>datetime.time(hour, minute, second)</code>	Створює об'єкт часу
<code>strftime(format)</code>	Форматує дату та час у рядок відповідно до заданого шаблону
<code>strptime(string, format)</code>	Перетворює рядок на об'єкт дати та часу
<code>timedelta(days, hours, minutes, seconds)</code>	Створює часовий інтервал для виконання арифметичних операцій з датою та часом

Окрім модулів *math*, *random* та *datetime*, які широко використовуються під час розроблення програм, стандартна бібліотека *Python* містить багато інших модулів, призначених для вирішення спеціалізованих завдань. Вони дозволяють працювати з текстовими даними, файлами та каталогами, виконувати математичні обчислення, взаємодіяти з операційною системою, використовувати функціональний стиль програмування, а також створювати графічні застосунки та реалізовувати інші можливості. Базові модулі стандартної бібліотеки *Python* та їх призначення наведені в таблиці 6.4.

Таблиця 6.4 – Базові модулі стандартної бібліотеки *Python*

Модуль	Призначення
1	2
Для роботи з текстом	
<i>string</i>	Загальні операції з рядками та форматування тексту
<i>re</i>	Робота з регулярними виразами
<i>difflib</i>	Порівняння текстових послідовностей і файлів
<i>unicodedata</i>	Робота із символами <i>Unicode</i>
<i>textwrap</i>	Форматування та перенесення тексту
Для роботи з різними типами даних	
<i>calendar</i>	Робота з календарями
<i>collections</i>	Спеціалізовані структури даних
<i>array</i>	Масиви однотипних елементів
<i>copy</i>	Поверхнєве та глибоке копіювання об'єктів

Продовження таблиці 6.4

1	2
Математичні модулі	
<i>numbers</i>	Абстрактні числові класи
<i>cmath</i>	Математичні функції для комплексних чисел
<i>decimal</i>	Десяткова арифметика підвищеної точності
<i>statistics</i>	Статистичні обчислення
<i>fractions</i>	Робота з раціональними числами
Для роботи з файлами та каталогами	
<i>pathlib</i>	Об'єктно-орієнтована робота зі шляхами
<i>os.path</i>	Маніпулювання шляхами до файлів
<i>fileinput</i>	Опрацювання кількох вхідних файлів
<i>stat</i>	Отримання інформації про файли
<i>filecmp</i>	Порівняння файлів і каталогів
<i>tempfile</i>	Створення тимчасових файлів і каталогів
<i>glob</i>	Пошук файлів за шаблоном
<i>fnmatch</i>	Порівняння імен файлів із шаблонами
<i>linecache</i>	Доступ до окремих рядків текстових файлів
<i>shutil</i>	Файлові операції високого рівня
Для підтримки функціонального програмування	
<i>itertools</i>	Засоби для ефективної роботи з ітераторами
<i>functools</i>	Функції вищого порядку та допоміжні інструменти
<i>operator</i>	Функціональні аналоги стандартних операторів
Для взаємодії з операційною системою	
<i>os</i>	Робота з операційною системою
<i>io</i>	Робота з потоками введення та виведення
<i>time</i>	Робота з часом
<i>argparse</i>	Опрацювання аргументів командного рядка
<i>getopt</i>	Альтернативне опрацювання параметрів командного рядка
<i>logging</i>	Ведення журналу подій програми
<i>getpass</i>	Безпечне введення паролів
<i>platform</i>	Отримання інформації про операційну систему
<i>errno</i>	Коди системних помилок
<i>ctypes</i>	Виклик функцій бібліотек, написаних іншими мовами
Сервіси середовища виконання	
<i>sys</i>	Системні параметри та функції <i>Python</i>
<i>sysconfig</i>	Інформація про конфігурацію <i>Python</i>
<i>builtins</i>	Доступ до вбудованих об'єктів
<i>__main__</i>	Організація виконання головного модуля

Продовження таблиці 6.4

1	2
<i>dataclasses</i>	Створення класів даних
<i>abc</i>	Абстрактні базові класи
<i>traceback</i>	Отримання та виведення трасування помилок
Для створення програмних застосунків	
<i>turtle</i>	Побудова графічних зображень засобами «черепашки»
<i>cmd</i>	Створення командних інтерпретаторів
<i>shlex</i>	Лексичний аналіз командних рядків

6.3 Сторонні (зовнішні) бібліотеки

Крім стандартної бібліотеки, яка входить до складу *Python*, існує багато сторонніх (зовнішніх) бібліотек, що значно розширюють можливості мови програмування. Серед різноманітних сторонніх бібліотек *Python* розглянемо найбільш поширені. Одними з найпопулярніших сторонніх бібліотек *Python* є бібліотеки *NumPy*, *Pandas*, *Matplotlib*.

NumPy (*Numerical Python*) – це бібліотека, яка призначена для ефективної роботи з багатовимірними масивами та виконання наукових і математичних обчислень. Вона є базовою бібліотекою для аналізу даних, машинного навчання, комп’ютерного моделювання та чисельних розрахунків тощо.

Основною структурою даних у *NumPy* є масив (це впорядкована, індексована структура даних, призначена для зберігання множини елементів одного типу (переважно числових), що розміщуються у суміжних комірках пам’яті), який, на відміну від стандартних списків *Python*, забезпечує значно швидшу обробку даних завдяки оптимізованій роботі на рівні низькорівневих операцій.

NumPy дозволяє виконувати векторизовані операції з масивами без використання циклів, що значно підвищує продуктивність програм. Головні можливості бібліотеки такі:

- створення масивів різної розмірності;

- виконання математичних операцій з масивами;
- лінійна алгебра (матриці, визначники, обернені матриці);
- статистичні обчислення;
- генерація випадкових чисел;
- робота з математичними функціями.

```
import numpy

# 1. Створення одновимірного масиву (вектора)
vector = numpy.array([1, 2, 3, 4, 5])
print("Вектор:", vector)
# Результат: Вектор: [1 2 3 4 5]

# 2. Векторизована операція (множення всього масиву на число)
# У звичайному Python знадобився б цикл, а в NumPy це робиться в один рядок:
doubled_vector = vector * 2
print("Подвоєний вектор:", doubled_vector)
# Результат: Подвоєний вектор: [2 4 6 8 10]

# 3. Створення двовимірного масиву (матриці 2x3)
matrix = numpy.array([[1, 2, 3], [4, 5, 6]])
print("Матриця:\n", matrix)
# Результат:
# Матриця:
# [[1 2 3]
#  [4 5 6]]

# 4. Швидке отримання властивостей матриці
print("Форма (рядків, колонок):", matrix.shape) # Результат: (2, 3)
print("Загальна кількість елементів:", matrix.size) # Результат: 6

# 5. Математичні обчислення по всьому масиву
print("Сума всіх елементів:", matrix.sum()) # Результат: 21
print("Середнє значення:", matrix.mean()) # Результат: 3.5
```

Рисунок 6.8 – Приклад використання бібліотеки *NumPy*

Бібліотека *NumPy* містить багато функцій (методів) для роботи з багатовимірними масивами, математичних обчислень та лінійної алгебри. Найбільш використовувані з них можна поділити на кілька груп (табл. 6.5).

Таблиця 6.5 – Найбільш використовувані функцій (методів) *NumPy*

Функція (метод)	Призначення
Створення масивів	
<i>np.array()</i>	Створення масиву зі списку або кортежу
<i>np.zeros()</i>	Масив, заповнений нулями
<i>np.ones()</i>	Масив, заповнений одиницями
<i>np.empty()</i>	Створення масиву без ініціалізації значень
<i>np.arange()</i>	Масив із заданим кроком
<i>np.linspace()</i>	Рівномірно розподілені значення
Робота з формою масиву	
<i>reshape()</i>	Зміна форми масиву
<i>ravel()</i>	Перетворення на 1D-масив
<i>transpose()</i>	Транспонування масиву
<i>shape</i>	Розмірність масиву
<i>size</i>	Кількість елементів
<i>dtype</i>	Тип даних елементів
Математичні операції	
<i>np.sum()</i>	Сума елементів
<i>np.mean()</i>	Середнє значення
<i>np.median()</i>	Медіана
<i>np.min()</i> / <i>np.max()</i>	Мінімальне / максимальне значення
<i>np.std()</i>	Стандартне відхилення
<i>np.var()</i>	Дисперсія
<i>np.abs()</i>	Модуль чисел
Лінійна алгебра	
<i>np.dot()</i>	Множення матриць / скалярний добуток
<i>np.linalg.inv()</i>	Обернена матриця
<i>np.linalg.det()</i>	Визначник матриці
<i>np.linalg.eig()</i>	Власні значення і вектори
<i>np.linalg.solve()</i>	Розв'язання систем лінійних рівнянь
Випадкові числа	
<i>np.random.rand()</i>	Випадкові числа (0–1)
<i>np.random.randn()</i>	Нормальний розподіл
<i>np.random.randint()</i>	Випадкові цілі числа
<i>np.random.choice()</i>	Випадковий вибір елементів
<i>np.random.shuffle()</i>	Перемішування масиву

Pandas (Python Data Analysis Library) – це бібліотека, яка призначена для зручної та ефективної обробки, аналізу та маніпулювання структурованими даними. Вона є однією з головних бібліотек у сфері аналізу

даних, машинного навчання, статистики та обробки великих масивів інформації.

Базовою структурою даних у *Pandas* є *Series* (одновимірний масив із підписаними індексами) та *DataFrame* (двовимірна таблична структура даних з рядками та стовпцями). *DataFrame* можна розглядати як таблицю, подібну до електронної таблиці *Excel* або *SQL*-таблиці, де кожен стовпець може мати свій тип даних.

На відміну від звичайних списків або масивів *Python*, *Pandas* забезпечує значно зручніші інструменти для роботи з табличними даними, включаючи фільтрацію, групування, агрегацію та обробку пропущених значень.

Pandas дозволяє виконувати складні операції з даними без необхідності написання низькорівневих циклів, що робить код більш зрозумілим та ефективним. Серед основних можливостей бібліотеки (табл. 6.6) – завантаження та збереження даних з різних форматів (*CSV*, *Excel*, *JSON* тощо); обробка та очищення даних; фільтрація, сортування та індексація даних; групування та агрегація даних; робота з пропущеними значеннями (*NaN*); об'єднання та злиття наборів даних; статистичний аналіз та базова візуалізація.

```
import pandas

# 1. Створення таблиці (DataFrame) на основі словника
data = {
    "Бренд": ["iGuzzini", "Philips", "Eaton", "Osram"],
    "Тип обладнання": ["Світильник", "Лампа", "Автомат", "Драйвер"],
    "Кількість": [12, 50, 8, 25],
    "Ціна, €": [120.5, 15.0, 45.0, 30.2]
}

df = pandas.DataFrame(data)
print("--- Повна таблиця ---")
print(df)

# 2. Фільтрація даних (вибираємо тільки позиції, де кількість > 15)
filtered_df = df[df["Кількість"] > 15]
print("\n--- Фільтрація (Кількість > 15) ---")
print(filtered_df)

# 3. Математичні операції зі стовпцями (додаємо новий стовпець «Вартість»)
df["Загальна вартість, €"] = df["Кількість"] * df["Ціна, €"]
print("\n--- Таблиця з розрахунком вартості ---")
print(df)

# 4. Швидка статистика по числових стовпцях
print("\n--- Загальна сума замовлення, € ---")
print(df["Загальна вартість, €"].sum()) # Результат: 3302.0
```

Рисунок 6.9 – Приклад використання бібліотеки *Pandas*

Таблиця 6.6 – Найбільш використовувані функцій (методів) *Pandas*

Функція (метод)	Призначення
Завантаження та збереження даних	
<i>pd.read_csv()</i>	Завантаження даних із <i>CSV</i> -файлу
<i>pd.read_excel()</i>	Завантаження даних з <i>Excel</i> -файлу
<i>pd.read_json()</i>	Завантаження даних із <i>JSON</i> -файлу
<i>df.to_csv()</i>	Збереження даних у <i>CSV</i>
<i>df.to_excel()</i>	Збереження даних в <i>Excel</i>
Перегляд даних	
<i>df.head()</i>	Перегляд перших рядків таблиці
<i>df.tail()</i>	Перегляд останніх рядків таблиці
<i>df.info()</i>	Загальна інформація про <i>DataFrame</i>
<i>df.describe()</i>	Статистичний опис даних
<i>df.shape</i>	Розмір таблиці (рядки, стовпці)
Робота з стовпцями та рядками	
<i>df['column']</i>	Вибір одного стовпця
<i>df.loc[]</i>	Доступ за мітками (іменами)
<i>df.iloc[]</i>	Доступ за індексами
<i>df.drop()</i>	Видалення рядків або стовпців
<i>df.rename()</i>	Перейменування стовпців
Фільтрація даних	
<i>df.query()</i>	Фільтрація за умовою
<i>df.isnull()</i>	Пошук пропущених значень
<i>df.notnull()</i>	Перевірка наявності значень
Очищення даних	
<i>df.fillna()</i>	Заповнення пропущених значень
<i>df.dropna()</i>	Видалення пропущених значень
<i>df.duplicated()</i>	Пошук дублікатів
<i>df.drop_duplicates()</i>	Видалення дублікатів
Агрегація та групування	
<i>df.groupby()</i>	Групування даних
<i>df.agg()</i>	Агрегація даних
<i>df.sum()</i>	Обчислення суми
<i>df.mean()</i>	Обчислення середнього значення
<i>df.count()</i>	Підрахунок значень
Об'єднання даних	
<i>pd.concat()</i>	Об'єднання таблиць
<i>pd.merge()</i>	Злиття таблиць
<i>df.join()</i>	Об'єднання по індексу

Matplotlib – це бібліотека *Python*, яка призначена для створення різноманітних графіків, діаграм і візуалізацій даних. *Matplotlib* дозволяє будувати як прості графіки (лінійні, стовпчикові, точкові), так і складні багатокомпонентні візуалізації з налаштуванням осей, підписів, кольорів, легенд та стилів оформлення. *Matplotlib* забезпечує високу гнучкість у налаштуванні візуалізацій, що дозволяє використовувати її як для навчальних цілей, так і для професійного аналізу даних.

Серед головних можливостей бібліотеки *Matplotlib* (табл. 6.7) – побудова лінійних графіків; створення стовпчикових діаграм; побудова кругових діаграм; створення точкових графіків; налаштування підписів осей та заголовків; робота з легендами та стилями; створення багатопанельних графіків; збереження графіків у різних форматах.

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

# 1. Підготовка даних (вісь X та вісь Y)
# Створюємо 100 точок від 0 до 2*pi
x = np.linspace(0, 2 * np.pi, 100)
# Обчислюємо синус для кожної точки X
y = np.sin(x)

# 2. Створення графіка
# figsize задає розмір вікна в дюймах (ширина, висота)
plt.figure(figsize=(8, 5))

# Побудова лінії (аргументи: x, y, колір, товщина, назва для легенди)
plt.plot(x, y, color='#1f77b4', linewidth=2.5, label='Графік синусоїди: sin(x)')

# 3. Оформлення
plt.title('Аналіз тригонометричної функції', fontsize=16, fontweight='bold')
plt.xlabel('Значення кута (в радіанах)', fontsize=12)
plt.ylabel('Амплітуда (sin)', fontsize=12)

# Додавання сітки для зручності читання
plt.grid(True, linestyle='--', alpha=0.7)

# Додавання легенди (показує 'label' з plt.plot)
plt.legend(loc='upper right', fontsize=10)

# 4. Відображення результату
plt.show()
```

Рисунок 6.10 – Приклад використання бібліотеки *Matplotlib*

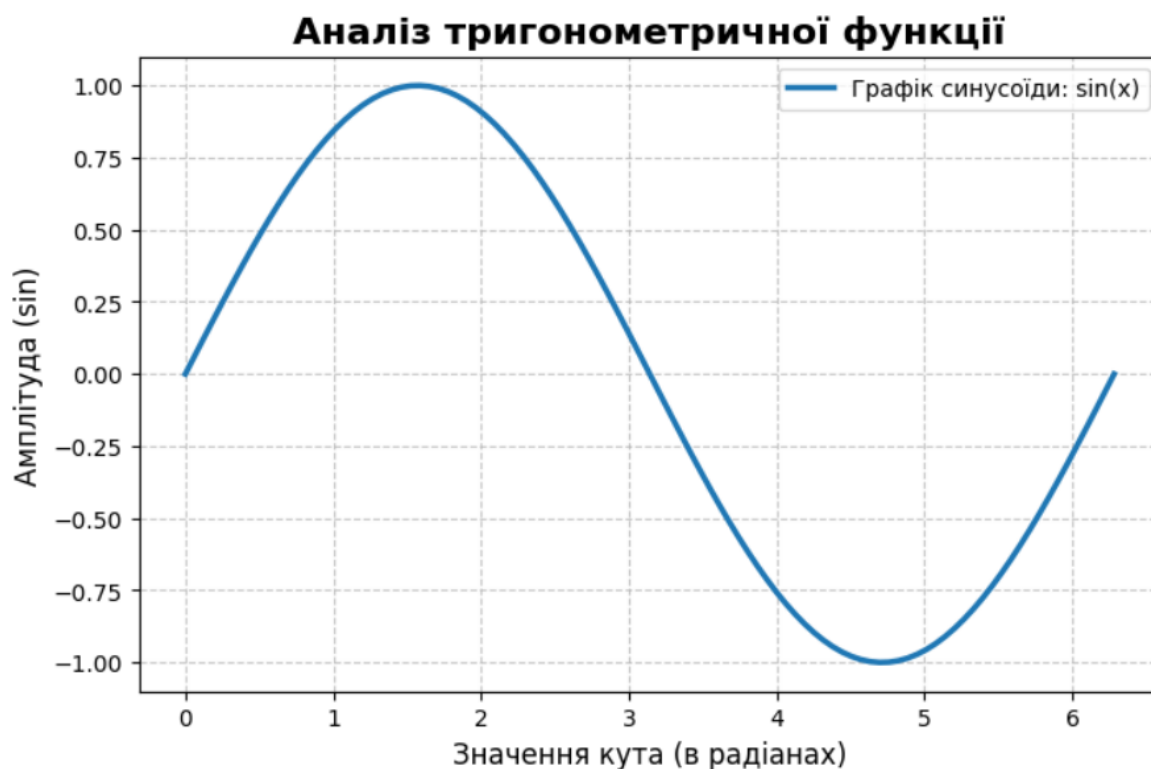


Рисунок 6.11 – Результат виконання коду, представленого на рисунку 6.9

Таблиця 6.7 – Найбільш використовувані функцій (методів) *Matplotlib*

Функція (метод)	Призначення
1	2
Створення графіків	
<code>plt.plot()</code>	Побудова лінійного графіка
<code>plt.scatter()</code>	Побудова точкового графіка
<code>plt.bar()</code>	Побудова стовпчикової діаграми
<code>plt.barh()</code>	Побудова горизонтальної стовпчикової діаграми
<code>plt.hist()</code>	Побудова гістограми
<code>plt.pie()</code>	Побудова кругової діаграми
Оформлення графіків	
<code>plt.title()</code>	Додавання заголовка графіка
<code>plt.xlabel()</code>	Підпис осі X
<code>plt.ylabel()</code>	Підпис осі Y
<code>plt.legend()</code>	Додавання легенди
<code>plt.grid()</code>	Відображення сітки
Керування областю побудови	
<code>plt.figure()</code>	Створення нового полотна (графіка)
<code>plt.subplot()</code>	Створення підграфіків
<code>plt.subplots()</code>	Створення сітки графіків

Продовження таблиці 6.7

1	2
Налаштування осей	
<code>plt.xlim()</code>	Встановлення меж осі X
<code>plt.ylim()</code>	Встановлення меж осі Y
<code>plt.xticks()</code>	Налаштування поділок осі X
<code>plt.yticks()</code>	Налаштування поділок осі Y
Збереження та відображення	
<code>plt.show()</code>	Відображення графіка
<code>plt.savefig()</code>	Збереження графіка у файл
Робота з об'єктами	
<code>fig, ax = plt.subplots()</code>	Створення об'єктів фігури та осей
<code>ax.plot()</code>	Побудова графіка в об'єктному стилі
<code>ax.set_title()</code>	Заголовок у об'єктному стилі
<code>ax.set_xlabel()</code>	Підпис осі X (об'єктний стиль)
<code>ax.set_ylabel()</code>	Підпис осі Y (об'єктний стиль)

Нижче наведено огляд інших популярних бібліотек *Python*, які зазвичай використовуються у практичних проєктах та допомагають вирішувати широкий спектр прикладних завдань.

Seaborn – це бібліотека, побудована на базі *Matplotlib*, яка призначена для створення статистичних візуалізацій даних у більш привабливому та зручному вигляді. Вона дозволяє будувати складні графіки з мінімальною кількістю коду та автоматично поліпшує їх оформлення.

Pillow (PIL) – бібліотека для роботи із зображеннями, що є розширенням *Python Imaging Library*. Вона надає інструменти для відкриття, редагування, конвертації форматів, зміни розміру, застосування фільтрів та обробки кольорів зображень.

Plotly та *Bokeh* – сучасні бібліотеки для створення інтерактивних графіків і дашбордів. Вони дозволяють будувати динамічні візуалізації, які можна використовувати у вебзастосунках та аналітичних панелях.

Бібліотеки для створення ігор

Pygame – найпопулярніша бібліотека для розробки 2D-ігор. Вона забезпечує інструменти для роботи з графікою, звуком, обробкою клавіатури, миші та інших пристроїв введення.

Pyglet – бібліотека для створення 2D- та простих 3D-ігор із підтримкою *OpenGL*. Вона не потребує додаткових залежностей і дозволяє працювати напряму з графічною системою.

PyOpenGL – бібліотека для створення 3D-графіки та ігор. Вона надає доступ до функцій *OpenGL*, що дозволяє моделювати тривимірні сцени, об'єкти, текстури та ефекти освітлення.

Бібліотеки машинного навчання та III

Scikit-learn – бібліотека для класичного машинного навчання. Вона містить алгоритми класифікації, регресії, кластеризації, а також інструменти для підготовки та аналізу даних.

PyTorch – бібліотека для розробки та навчання моделей штучного інтелекту. Вона підтримує динамічні обчислювальні графи та широко використовується у дослідженнях і нейронних мережах.

TensorFlow – потужна платформа для глибокого навчання. Вона дозволяє створювати складні нейронні мережі та працювати з великими обсягами даних для завдань розпізнавання зображень і мови.

Веброзробка та автоматизація

Selenium – бібліотека для автоматизації веббраузера. Вона дозволяє програмно виконувати дії на вебсторінках: натискання кнопок, заповнення форм та навігацію.

Beautiful Soup – бібліотека для парсингу *HTML*- і *XML*-документів. Вона використовується для вилучення та обробки даних із вебсторінок.

Вебфреймворки

Flask – легкий мікрофреймворк для створення вебзастосунків. Він простий у використанні та зазвичай застосовується для невеликих проєктів і *REST API*.

Django – повнофункціональний вебфреймворк із принципом «все включено». Він має вбудовані інструменти для роботи з базами даних, автентифікацією та адміністративною панеллю, що робить його зручним для використання у великих проєктах.

7 РОБОТА З ФАЙЛАМИ. ВИНЯТКИ

7.1 Основні операції з файлами

Під час виконання програми всі дані тимчасово зберігаються в оперативній пам'яті (*Random Access Memory, RAM*). Після завершення роботи програми або вимкнення комп'ютера інформація, що міститься в оперативній пам'яті, втрачається. Тому для довготривалого зберігання даних використовують файли, які розміщуються на зовнішніх носіях інформації, таких як жорсткі диски, SSD-накопичувачі, флеш-пам'ять тощо.

Файл становить іменовану область зовнішньої пам'яті, призначену для постійного зберігання інформації. Для зручності великі набори файлів організовують у директорії (папки). Кожний файл має унікальне ім'я, а його повне розташування визначається шляхом, що складається з імен директорій та імені файлу.

Робота з файлами дозволяє програмам зберігати результати обчислень, обмінюватися інформацією між різними застосунками, опрацьовувати великі обсяги даних, а також створювати документи різних форматів, наприклад текстові файли, електронні таблиці або *pdf*-документи.

У *Python* працюють здебільшого з текстовими файлами, які містять символічну інформацію, та бінарними файлами, що зберігають дані у двійковому форматі (зображення, аудіо, відео, архіви тощо).

Принцип роботи з файлами значною мірою нагадує роботу з книгою. Щоб прочитати книгу, її спочатку необхідно відкрити, після завершення читання – закрити. Аналогічно і під час роботи з файлами: перед виконанням будь-яких операцій файл необхідно відкрити, після завершення роботи – закрити. Поки файл відкритий, із нього можна отримувати інформацію або записувати до нього нові дані. При цьому система відстежує поточну позицію у файлі, що дає змогу читати або записувати інформацію послідовно чи переходити до потрібного місця.

У *Python* відкриття файлу здійснюється за допомогою вбудованої функції *open()*, яка повертає файловий об'єкт. Саме через цей об'єкт виконуються всі подальші операції читання, запису та керування файлом.

У загальному вигляді відкриття файлу виконується так:

ім'я_об'єкта = open("ім'я_файлу", "режим"),

де ім'я_об'єкта – змінна, у якій зберігається файловий об'єкт; ім'я_файлу – назва або повний шлях до файлу; режим – спосіб відкриття файлу (читання, запис, дозапис тощо).

Під час роботи з файлами в *Python* можна виокремити чотири головні операції: відкриття, читання, запис та закриття файлу. Кожна з цих операцій виконує певну функцію та є важливою складовою процесу опрацювання файлових даних:

- відкриття файлу – надання програмі доступу до вмісту файлу для подальшого читання, запису або інших операцій.
- читання файлу – отримання інформації з файлу та передача її програмі для подальшого опрацювання;
- запис у файл – створення нового вмісту файлу або зміна існуючого шляхом додавання, заміни чи перезапису даних;
- закриття файлу – завершення роботи з файлом, збереження внесених змін (за потреби) та звільнення системних ресурсів, пов'язаних із відкритим файлом;

Для виконання операцій з файлом недостатньо лише вказати його ім'я. Під час відкриття необхідно також визначити режим доступу, який встановлює спосіб роботи з файлом. Режим відкриття визначає, чи буде файл використовуватися для читання, запису, дозапису, створення нового файлу або комбінованого читання і запису. Крім того, він дозволяє обрати текстовий або бінарний формат роботи з даними.

Головні режими відкриття файлів у *Python* наведено в таблиці 7.1.

Таблиця 7.1 – Головні режими відкриття файлів

Режим	Призначення
<i>r</i>	Відкриває файл лише для читання. Якщо файл не існує, виникає помилка <i>FileNotFoundError</i> . Це режим відкриття за замовчуванням
<i>w</i>	Відкриває файл для запису. Якщо файл існує, його вміст повністю видаляється. Якщо файл відсутній, він буде створений
<i>a</i>	Відкриває файл для дозапису. Нові дані додаються в кінець файлу без видалення наявного вмісту. Якщо файл не існує, він буде створений
<i>x</i>	Створює новий файл для запису. Якщо файл із таким ім'ям уже існує, виникає помилка <i>FileExistsError</i>
<i>t</i>	Відкриває файл у текстовому режимі. Використовується за замовчуванням і може поєднуватися з іншими режимами (<i>rt</i> , <i>wt</i> , <i>at</i>)
<i>b</i>	Відкриває файл у бінарному режимі. Використовується для роботи із зображеннями, аудіо, відео, архівами та іншими двійковими даними (<i>rb</i> , <i>wb</i> , <i>ab</i>)
<i>+</i>	Дозволяє одночасно читати та записувати файл. Використовується разом з іншими режимами (<i>r+</i> , <i>w+</i> , <i>a+</i>)

Варто зазначити, що режими відкриття файлів можна комбінувати між собою. Наприклад, режим *rb* відкриває файл для читання у бінарному форматі, *w+* – для одночасного читання та запису з очищенням вмісту файлу, а *a+* – для читання та дозапису даних у кінець файлу.

Для читання інформації файл необхідно спочатку відкрити в режимі читання (*r*), вказавши його ім'я або шлях за допомогою функції *open()*. Після відкриття створюється файловий об'єкт, через який програма отримує доступ до вмісту файлу. Далі для зчитування даних використовуються такі спеціальні методи: *read()* – для читання всього вмісту файлу, *readline()* – для читання одного рядка, *readlines()* – для зчитування всіх рядків у вигляді списку. Після завершення роботи файл необхідно закрити методом *close()* або використати конструкцію *with*, яка виконує це автоматично.

Для запису інформації файл необхідно відкрити у відповідному режимі: *w*, *a* або *w+* залежно від способу збереження даних. Режим *w* використовується для створення нового файлу або повного перезапису існуючого, *a* – для додавання нових даних у кінець файлу без зміни його

вмісту, а w^+ – для одночасного читання та запису з очищенням попереднього вмісту файлу. Після відкриття створюється файловий об’єкт, через який запис даних здійснюється за допомогою методів *write()* або *writelines()*. Після завершення запису файл необхідно закрити або використати конструкцію *with*, яка автоматично завершує роботу з файлом.

Після завершення роботи з файлом його необхідно закрити, щоб звільнити системні ресурси та забезпечити коректне завершення операцій читання або запису. Закриття файлу виконується за допомогою методу *close()*, який завершує доступ програми до файлового об’єкта.

Загальний вигляд закриття файлу є таким:

file.close()

Після закриття файлу будь-які спроби читання або запису через цей файловий об’єкт призведуть до виникнення помилки. У сучасному програмуванні замість явного виклику методу *close()* рекомендується використовувати конструкцію *with*, яка автоматично закриває файл після завершення роботи з ним навіть у разі виникнення винятків. Це робить програму більш надійною та зменшує імовірність втрати даних або витоку системних ресурсів.

Методи для роботи з файлами подано в таблиці 7.2.

Таблиця 7.2 – Методи для роботи з файлами

Метод	Опис дії
1	2
<i>close()</i>	Закриває відкритий файл. Нічого не відбувається, якщо файл вже закритий
<i>detach()</i>	Відокремлює базовий двійковий буфер від <i>TextIOBase</i> та повертає його
<i>fileno()</i>	Повертає ціле число файлу (файловий дескриптор)
<i>flush()</i>	Очищує буфер запису у файловому потоці

Продовження таблиці 7.2

1	2
<i>isatty()</i>	Повертає <i>True</i> , якщо потік файлів є інтерактивним
<i>read(n)</i>	Зчитує не більше <i>n</i> символів з файлу. Виконує читання до кінця файлу, якщо <i>n</i> є від'ємним або <i>None</i>
<i>readable()</i>	Повертає <i>True</i> , якщо з файлового потоку можна зчитати дані
<i>readline(n=-1)</i>	Зчитує та повертає один рядок з файлу. Зчитує не більше <i>n</i> байтів, якщо це вказано
<i>readlines(n=-1)</i>	Зчитує та повертає список рядків з файлу. Зчитує не більше <i>n</i> байтів / символів, якщо це вказано
<i>seek(offset, from=SEEK_SET)</i>	Змінює позицію файлу на <i>offset</i> байтів відносно <i>from</i> (<i>start, current, end</i>)
<i>seekable()</i>	Повертає <i>True</i> , якщо файловий потік підтримує довільний доступ
<i>tell()</i>	Повертає ціле число, що становить поточну позицію файлового об'єкта
<i>truncate(size=None)</i>	Змінює розмір файлового потоку до <i>size</i> байтів. Якщо розмір не вказано, він змінюється щодо поточного розташування
<i>writable()</i>	Повертає <i>True</i> , якщо у файловий потік можна здійснити запис
<i>write(s)</i>	Записує рядок (<i>s</i>) у файл та повертає кількість записаних символів
<i>writelines(lines)</i>	Записує список рядків (<i>lines</i>) у файл

7.2 Винятки

Під час виконання програми можуть виникати різні непередбачені ситуації, які призводять до порушення її сталої роботи. Такі ситуації називаються винятками (*exceptions*). Виняток є спеціальним повідомленням

інтерпретатора *Python* про те, що під час виконання програми виникла помилка або інша подія, яка потребує обробки. Якщо в програмному коді не передбачено механізм обробки винятків, виконання програми буде перервано, а користувач отримає повідомлення про помилку.

Зазвичай поняття «помилка» (*error*) і «виняток» (*exception*) використовують як синоніми, однак між ними існує певна відмінність.

Помилка (Error) – це загальне поняття, що характеризує будь-яку некоректну ситуацію в програмі. До помилок належать синтаксичні, логічні помилки, несумісність бібліотек, нескінченна рекурсія та інші проблеми, які зазвичай призводять до аварійного завершення програми і не можуть бути оброблені стандартними засобами *Python*.

Виняток (Exception) – це спеціальний об'єкт, який виникає під час виконання програми та сигналізує про появу певної проблеми, наприклад, спробу поділити число на нуль, звернення до неіснуючого файлу або введення користувачем некоректних даних. На відміну від помилок, винятки можуть бути перехоплені та оброблені програмою за допомогою відповідних конструкцій.

Обробка винятків – це механізм, який дозволяє програмі виявляти, перехоплювати та опрацьовувати помилки, що виникають під час її виконання. Завдяки цьому програма не завершує роботу аварійно, а може виконати визначені дії: повідомити користувачеві про помилку, запропонувати повторне введення даних, записати інформацію про помилку до журналу або продовжити виконання програми.

Найпростішим механізмом обробки винятків у *Python* є конструкція *try...except*. Код, у якому потенційно може виникнути виняток, розміщується всередині блоку *try*. Якщо під час його виконання виникає помилка, виконання блоку *try* припиняється, а керування передається до блоку *except*, у якому описуються дії для обробки винятку.

Загальна структура конструкції має такий вигляд:

try:

Код, у якому може виникнути виняток

except:

Код для обробки винятку

Конструкція *try...except...else* розширює можливості обробки винятків за рахунок використання блоку *else*. Код, який може згенерувати виняток, розміщується в блоці *try*, а його обробка виконується в блоці *except*. Якщо під час виконання блоку *try* не виникло жодного винятку, автоматично виконується блок *else*. Це дозволяє відокремити основну логіку програми від коду обробки помилок, що робить програму більш зрозумілою та структурованою.

Загальна структура конструкції має такий вигляд:

try:

Код, у якому може виникнути виняток

except:

Обробка винятку

else:

Код, який виконується, якщо виняток не виник

Конструкція *try...finally* використовується тоді, коли необхідно обов'язково виконати певний код незалежно від того, чи виник виняток під час виконання програми. Код, який може згенерувати виняток, розміщується в блоці *try*, а блок *finally* виконується завжди – як при успішному завершенні блоку *try*, так і в разі виникнення винятку.

Здебільшого блок *finally* застосовують для закриття файлів, звільнення системних ресурсів, завершення мережових з'єднань або виконання інших дій, які повинні бути виконані незалежно від результату роботи програми.

Загальна структура конструкції має такий вигляд:

try:

Код, у якому може виникнути виняток

finally:

Код, який виконується за будь-яких умов

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Костюченко А. О. Основи програмування мовою Python : навч. посіб. / А. О. Костюченко ; Нац. ун-т «Чернігівський колегіум» імені Т. Г. Шевченка. – Чернігів : ФОП Баликіна С. М., 2020. – 180 с.
2. Anaya M. Clean Code in Python : Refactor your legacy code base. – Birmingham : Packt Publishing, 2018. – 328 p.
3. Introduction to Python Programming / U. Das, A. Lawson, C. Mayfield, N. Norouzi. – Houston : OpenStax, Rice University, 2024. – 414 p.
4. Dr. Pooja. Data Visualization with Python: Exploring Matplotlib, Seaborn, and Bokeh for Interactive Visualizations / Dr. Pooja. – London : BPB Online, 2023. – 284 p.
5. Matthees E. Python crash course: a hands-on, project-based introduction to programming / E. Matthees – 2nd ed. – San Francisco : No Starch Press, 2019. – 544 p.
6. Lutz M. Learning Python: Powerful Object-Oriented Programming / M. Lutz. – 6th ed. – Sebastopol, CA : O'Reilly Media, 2025. – 1230 p.
7. Guta G. Pragmatic Python Programming: Learning Python the Smart Way / G. Guta. – Budapest, Hungary : Apress, 2022. – 198 p.
8. Програмування числових методів мовою Python : підручник / А. В. Анісімов та ін. ; за ред. А. В. Анісімова. – Київ : Видавничо-поліграфічний центр «Київський університет», 2014. – 640 с.
9. Васильєв О. Програмування в Python. Теорія і практика : навч. посіб. / О. Васильєв. – Київ : Видавництво Ліра-К, 2023. – 462 с.
10. Яковенко А. Основи програмування. Python. Частина 1 : підручник для студентів спеціальності 122 «Комп'ютерні науки», спеціалізації «Інформаційні технології в біології та медицині» / А. Яковенко ; КПІ ім. Ігоря Сікорського. – Київ : КПІ ім. Ігоря Сікорського, 2018. – 195 с.

Електронне навчальне видання

ВОРОНОВ Роман Володимирович,
ГЕРАСИМЕНКО Віталій Анатолійович

ОСНОВИ ПРОГРАМУВАННЯ

КОНСПЕКТ ЛЕКЦІЙ

(для здобувачів першого (бакалаврського) рівня вищої освіти денної та заочної форм навчання зі спеціальності ГЗ – Електрична інженерія)

Відповідальний за випуск *В. А. Герасименко*

Редактор *О. А. Норик*

Комп'ютерне верстання *Р. В. Воронов, І. В. Волосожарова*

План 2026, поз. 82Л

Підп. до друку 01.09.2026. Формат 60 × 84/16.

Ум. друк. арк. 6,1.

Видавець і виготовлювач:

Харківський національний університет
міського господарства імені О. М. Бекетова,
вул. Чорноглазівська, 17, Харків, 61002.

Електронна адреса: office@kname.edu.ua

Свідоцтво суб'єкта видавничої справи:

ДК № 8386 від 14.07.2025.