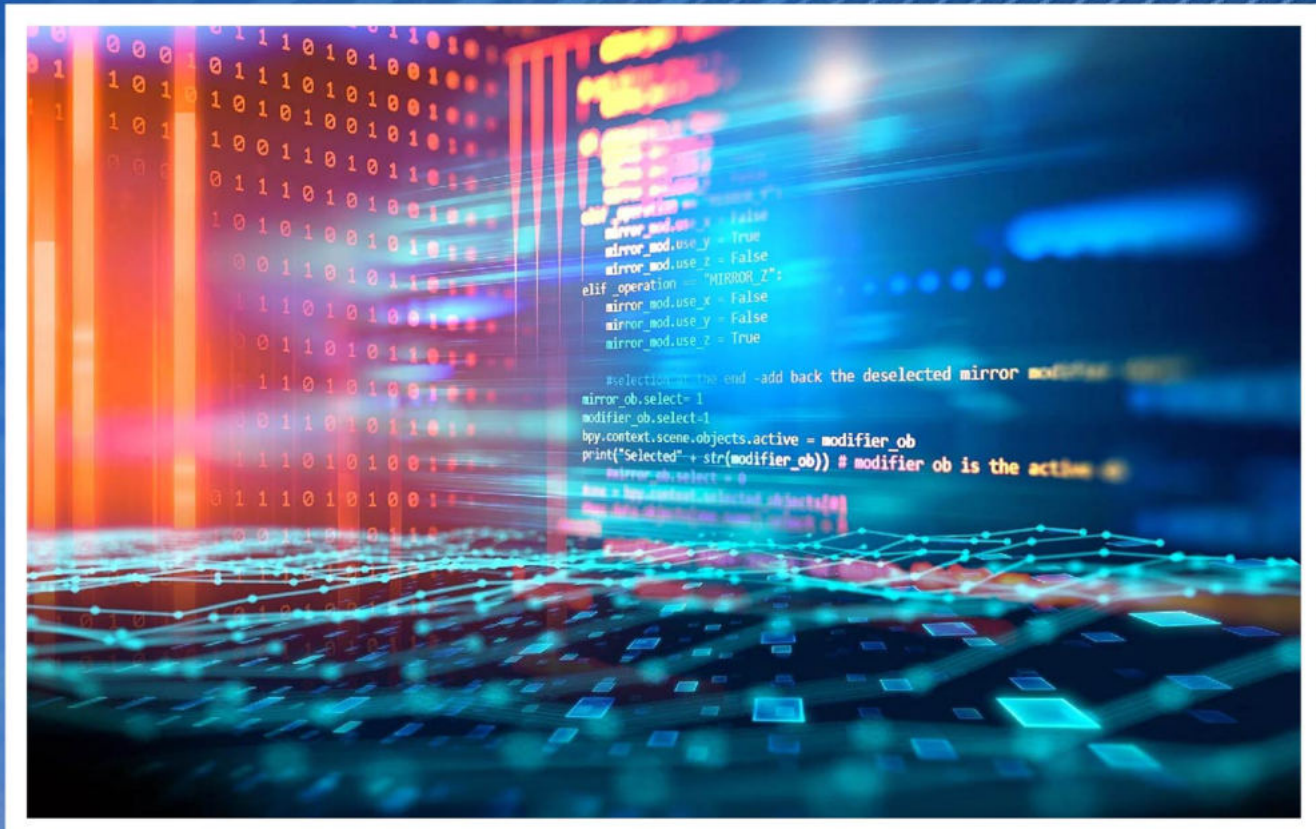
В. В. Лаговський,
М. М. Філоненко, С. М. Грищенко

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ



НАВЧАЛЬНИЙ ПОСІБНИК

ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ

ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Навчальний посібник

Ірпінь
2024

УДК 004.43(075.8)
ББК 32.973.2я73
Л 14

Рекомендовано до друку Вченою радою
Державного податкового університету
(протокол № 6 від 18 листопада 2024 року)

Рецензенти:

В. М. Франчук, д-р пед. наук, доцент, завідувач кафедра комп'ютерної та програмної інженерії Українського державного університету імені Михайла Драгоманова;

А. Ю. Горбовий, д-р тех. наук, професор, професор кафедри комп'ютерних та інформаційних технологій і систем Державного податкового університету.

Автори:

В. В. Лаговський, канд. екон. наук, доцент, доцент кафедри комп'ютерних та інформаційних технологій і систем; **М. М. Філоненко,** канд. фіз.-мат. наук, доцент, завідувач кафедри комп'ютерних та інформаційних технологій і систем; **С. М. Грищенко,** канд. пед. наук, старший дослідник, доцент кафедри комп'ютерних та інформаційних технологій і систем.

Лаговський В. В.

Л 14 **Об'єктно-орієнтоване програмування** : навчальний посібник / В. В. Лаговський, М. М. Філоненко, С. М. Грищенко. – Ірпінь : Державний податковий університет, 2024. – 504 с.
ISBN 978-966-337-752-0

У навчальному посібнику викладено теоретичні та практичні засади об'єктно-орієнтованого програмування (ООП) з використанням мови Python. Розглянуто ключові концепції парадигми: класи та об'єкти, інкапсуляція, успадкування, поліморфізм та абстракція. Окрему увагу приділено особливостям реалізації ООП у Python, зокрема роботі з «магічними» методами, декораторами та винятками.

Посібник спрямований на формування професійних компетентностей у сфері розробки програмного забезпечення. Видання призначене для здобувачів першого (бакалаврського) рівня вищої освіти за спеціальностями F2 «Інженерія програмного забезпечення» та F3 «Комп'ютерні науки», а також усіх, хто прагне поглибити свої знання у Python-розробці.

УДК 004.43(075.8)
ББК 32.973.2я73

ISBN 978-966-337-752-0

ЗМІСТ

ЗМІСТОВИЙ МОДУЛЬ 2
ОСНОВИ ТА ПРИНЦИПИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ НА ПРИКЛАДІ МОВИ PYTHON

ПЕРЕДМОВА

Дисципліна «Об'єктно-орієнтоване програмування» є однією з фундаментальних дисциплін підготовки бакалаврів за освітньо-професійною програмою «Технології цифрового дизайну» та «Інформаційні управляючі системи і технології (за галузями)» спе-ціальності 122 Комп'ютерні науки. Набуття вмінь та навичок з основ об'єктно-орієнтованого програмування є базою, що забезпечує подальше вивчення спеціальних дисциплін, пов'язаних з фаховою діяльністю.

Мета навчальної дисципліни: вивчення базових понять та принципів об'єктно-орієнтованого підходу до програмування і набуття навичок практичного застосування їх під час рішення типових задач програмування. Об'єктно-орієнтоване проєктування програмного забезпечення та реалізація його основних принципів вивчається на базі засобів мови Python.

Завдання навчальної дисципліни: ознайомлення з ефективними методами об'єктно-орієнтованого програмування, а також існуючим інструментарієм створення складних програмних систем для різноманітних предметних областей, сформувати у студентів знання та практичні навички об'єктно-орієнтованого програмування (ОПП) на основі мови Python та вміння застосовувати стандартні бібліотеки.

Посібник призначений для практичного опанування парадигми об'єктно-орієнтованого програмування із застосуванням мови Python. У ньому в систематизованому вигляді наводяться короткі теоретичні відомості, типові підходи проєктування об'єктно-орієнтованих проєктів та приклади розв'язання задач.

Теоретичний матеріал посібника пояснюється на великій кількості типових прикладів, частина з яких супроводжуються описами алгоритмів і поясненням ключових моментів програм (коду). Посібник складається з двох розділів, які містять 27 тем, кожна з яких присвячена певним аспектам об'єктно-орієнтованого програмування. Більшість тем поділені на пункти, нумерація яких містить номер теми, якій належить пункт.

У цьому посібнику надається всебічний огляд основ об’єктно-орієнтованого програмування (ООП) з використанням однієї з найпопулярніших і потужних мов програмування – Python.

Python завдяки своїй простоті та зручності стала популярною в багатьох розробників, починаючи від новачків до досвідчених професіоналів. Синтаксис мови Python зрозумілий і легкий для освоєння, а численні бібліотеки та фреймворки розширюють можливості для реалізації різноманітних завдань. Однак однією з найважливіших тем, яку потрібно вивчити для глибшого розуміння Python, є об’єктно-орієнтоване програмування.

Об’єктно-орієнтоване програмування – це підхід до програмування, який організовує програму як сукупність об’єктів, що взаємодіють між собою. Цей підхід дає змогу структурувати код так, щоб він був більш модульним, повторно використовуваним і легким для підтримки. Основні концепції ООП, як-от класи, об’єкти, наслідування та поліморфізм, допомагають розробникам створювати гнучкі й масштабовані програми.

У цьому посібнику розпочнемо з основ Python, що дасть змогу отримати чітке уявлення про мову та її можливості. У другому модулі заглибимося в об’єктно-орієнтоване програмування, розглядаючи ключові концепції та принципи, які допоможуть ефективно використовувати ці можливості для створення складних і ефективних програмних рішень.

Цей посібник стане корисним інструментом у навчанні та допоможе стати впевненим користувачем Python і об’єктно-орієнтованого програмування.

ЗМІСТОВНИЙ МОДУЛЬ 1
ОСНОВИ ПРОГРАМУВАННЯ НА ПРИКЛАДІ МОВИ PYTHON

Тема 1
Вступ до програмування

1.1. Загальне поняття про програмне забезпечення сучасних комп'ютерів.

1.2. Поняття про програмне забезпечення системного рівня.

1.3. Класифікація прикладного програмного забезпечення.

1.4. Алгоритмічні мови й системи програмування.

1.5. Рівні алгоритмічних мов програмування.

1.1. Загальне поняття про програмне забезпечення сучасних комп'ютерів

Програмне забезпечення сучасних комп'ютерів – це сукупність програм та інструментів, які використовуються для управління і виконання різних завдань на комп'ютері. Воно охоплює різноманітні програми, операційні системи, драйвери, бібліотеки та інше програмне забезпечення, необхідне для правильної роботи комп'ютерних систем.

Основні аспекти програмного забезпечення сучасних комп'ютерів:

1. Операційні системи (ОС). Операційні системи, такі як Windows, macOS, Linux, Android, iOS, є базовим програмним забезпеченням, яке керує ресурсами комп'ютера та забезпечує інтерфейс для взаємодії користувача з апаратним забезпеченням.

Основні функції ОС:

- Виконання на вимогу користувача низькорівневих дій, які є спільними для більшості програм і часто наявні майже в усіх програмах (введення та виведення даних, запуск і зупинка інших програм, виділення та вивільнення додаткової пам'яті тощо).
- Стандартизований доступ до периферійних пристроїв.

- Завантаження програм в оперативну пам’ять і їх виконання.
- Керування оперативною пам’яттю.
- Керування доступом до даних енергонезалежних носіїв.
- Відтворення інтерфейсу користувача.
- Мережеві операції, підтримка мережевих протоколів.

Додаткові функції ОС:

- Паралельне або псевдопаралельне виконання задач (багатозадачність).
- Розподіл ресурсів обчислювальної системи між процесами.
- Організація надійних обчислень (неможливості впливу процесу на перебіг інших), основана на розмежуванні доступу до ресурсів.
- Взаємодія між процесами: обмін даними, синхронізація.
- Захист самої системи, даних користувача і програм від дій користувача або інших програм.
- Багатокористувацький режим роботи та розподілення прав доступу.

Відносно свого призначення, операційні системи бувають:

- універсальні;
- спеціальні;
- спеціалізовані;
- однозадачні;
- багатозадачні;
- однокористувацькі;
- багатокористувацькі;
- реального часу.

Відносно способу інсталяції операційної системи їх поділяють на:

- вмонтовані (зберігаються в енергонезалежній пам’яті обчислювальної машини або пристрою без можливості заміни в процесі експлуатації обладнання);
- невмонтовані (інсталюються на один з пристроїв зберігання інформації обчислювальної машини з можливістю подальшої заміни в процесі експлуатації).

Відносно відповідності стандартам операційні системи бувають:

- стандартні (відповідають одному з загальноприйнятих відкритих стандартів, найчастіше POSIX);
- нестандартні.

Відносно ліцензії, можливостей розширення та можливостей внесення змін до вихідного коду операційні системи бувають:

- вільні – з вільними програмним кодом (GNU, BSD, MIT);
- відкриті (open source) – з відкритим програмним кодом;
- власницькі (proprietary) – комерційні з закритим кодом.

До складу операційної системи входять:

- ядро операційної системи, що забезпечує розподіл та керування ресурсами обчислювальної системи;
- базовий набір прикладних програм, системні бібліотеки та програми обслуговування.

Ядро системи – це набір функцій, структур даних та окремих програмних модулів, які завантажуються в пам’ять комп’ютера під час завантаження операційної системи та забезпечують три типи системних сервісів:

- керування введенням-виведенням інформації;
- керування оперативною;
- керування процесами.

Кожна з цих підсистем представлена відповідними функціями ядра системи.

Багатозадачні операційні системи також включають ще одну обов’язкову складову – механізм підтримки багатозадачності. Ця складова не надається як системний сервіс і тому не може бути віднесена до жодної з підсистем.

Три основних механізми забезпечення багатозадачності:

1) надання процесора окремій задачі на квант часу, який визначається самою задачею;

2) надання процесора окремій задачі на квант часу, який визначається обладнанням обчислювальної системи – інтервальним таймером;

3) виділення під окрему задачу окремого процесора в багатопроцесорних системах.

У перших двох випадках на кожному з процесорів в окремо взятий момент часу обраховується лише одна задача, але за рахунок достатньо малого кванта часу (в межах мілісекунд), що почергово надається кожній з задач, виникає ілюзія одночасного виконання в системі багатьох задач. У сучасних системах зазвичай комбінуються методи 2 і 3.

2. Програми для офісної роботи. Програмне забезпечення для офісної роботи, таке як Microsoft Office, LibreOffice, Google Workspace, надає інструменти для створення та редагування документів, таблиць, презентацій тощо.

3. Інтернет-браузери. Браузери, такі як Google Chrome, Mozilla Firefox, Microsoft Edge, Safari, слугують для перегляду вебсторінок та взаємодії з інтернетом.

Браузер під'єднується до сервера HTTP, отримує з нього документ і форматує його для представлення користувачеві або намагається викликати зовнішню програму, яка це зробить, залежно від формату документа. Формати документа, які вебпереглядач повинен представляти без допомоги зовнішніх програм, визначає World Wide Web Consortium (скорочено W3C). До них належать формати текстових документів HTML та XHTML, а також найпоширеніші формати растрової графіки GIF, JPEG та PNG (останній – розробка W3C). Якщо ви читаєте цей текст з екрана монітора, то в цей час ви користуєтесь вебпереглядачем.

Адресування сторінок відбувається за допомогою URL (Uniform Resource Locator, RFC 1738), який інтерпретується як адреса, що починається з *http:* для протоколу HTTP. Багато навігаторів також підтримують інші типи URL та їх відповідні протоколи, як, наприклад, *gopher:* для Gopher (ієрархічний протокол гіперпосилань), *ftp:* для протоколу перенесення файлів FTP, *rtsp:* для протоколу потоків реального часу RTSP, та *https:* для HTTPS (HTTP Secure, що розширює HTTP за допомогою Secure Sockets Layer SSL або Transport Layer Security TLS).

4. Розважальне програмне забезпечення. Містить ігри, мультимедійні програвачі, програми для стрімінгу відео та музики, редактори фотографій та відео.

5. Розробницькі інструменти. Включають інтегровані середовища розробки (IDE), текстові редактори, компілятори, інтерпретатори, версійні системи контролю, дебагери та інші інструменти, що використовуються для написання та відлагодження програм.

6. Бібліотеки та фреймворки. Це набори готових компонентів, які дають змогу розробникам прискорити розробку програм через використання вже написаного та перевіреного коду. Наприклад, TensorFlow та PyTorch для машинного навчання, Django та Flask для веброзробки.

7. Додатки для мобільних пристроїв. Це програмне забезпечення, призначене для використання на мобільних пристроях, таких як смартфони та планшети, охоплюючи ігри, соціальні мережі, месенджери та інші.

Програмне забезпечення сучасних комп'ютерів відіграє ключову роль у виконанні різноманітних завдань та забезпеченні зручної й продуктивної роботи користувачів.

1.2. Поняття про програмне забезпечення системного рівня

Програмне забезпечення системного рівня забезпечує базовий функціонал для операційних систем і дає змогу взаємодіяти з апаратним забезпеченням комп'ютера. Це охоплює драйвери, ядра операційних систем, системні бібліотеки та інші компоненти, які надають основні сервіси та ресурси для роботи комп'ютера. Ключові аспекти програмного забезпечення системного рівня:

1. Драйвери – програмне забезпечення, яке дає змогу операційній системі та програмам взаємодіяти з апаратним забезпеченням комп'ютера, таким як принтери, графічні картки, мережеві адаптери та інше. Вони перетворюють команди від операційної системи на інструкції, які зрозумілі для конкретного пристрою.

2. Ядро операційної системи –це центральний компонент операційної системи, який керує роботою апаратного забезпечення та надає основні сервіси, такі як керування процесами, планування завдань, керування пам'яттю та введенням-виведенням.

3. Системні бібліотеки – набори функцій і процедур, які надають доступ до базових операцій апаратного забезпечення та ресурсів комп'ютера. Вони використовуються програмами для взаємодії з операційною системою та апаратним забезпеченням.

4. Утиліти та інструменти командного рядка. Утиліти та інструменти командного рядка надають користувачам можливість взаємодіяти з операційною системою та виконувати різноманітні завдання, такі як керування файлами, адміністрування системи, моніторинг ресурсів та інше.

5. Бібліотеки для розробників. Бібліотеки для розробників надають доступ до додаткових можливостей та функцій для розробки програмного забезпечення, таких як робота з мережами, криптографія, обробка даних та інше.

6. Системні сервіси та служби. Системні сервіси та служби забезпечують роботу операційної системи та додатковий функціонал для користувачів і програм. Вони можуть охоплювати служби безпеки, мережеві сервіси, служби забезпечення цілісності даних та інше.

Ці компоненти програмного забезпечення системного рівня надають основні сервіси та ресурси, які потрібні для правильної роботи комп'ютерних систем і програм.

1.3. Класифікація прикладного програмного забезпечення

Прикладне програмне забезпечення (ППЗ) можна класифікувати за різними критеріями, такими як призначення, функціональність, сфера застосування та інші. Ось деякі загальні класифікаційні категорії прикладного програмного забезпечення:

1. Призначення застосування:

– офісні програми: Містять програми для створення та редагування текстових документів (наприклад, Microsoft Word, Google Docs), електронних таблиць (наприклад, Microsoft Excel, Google Sheets), презентацій (наприклад, Microsoft PowerPoint, Google Slides) та інші офісні інструменти;

– графічні та дизайнерські програми: Використовуються для розробки графічних дизайнів, мультимедійних вмістів, обробки фотографій, відеоредагування та іншого (наприклад, Adobe Photoshop, Adobe Illustrator, CorelDRAW);

– розрахунково-фінансове програмне забезпечення: Містить програми для обліку фінансів, ведення бухгалтерського обліку, розрахунків, податкового обліку та іншого (наприклад, QuickBooks, Xero, SAP);

– програмне забезпечення для управління відносинами з клієнтами (CRM): Використовується для автоматизації взаємодії з клієнтами, управління продажами, маркетингом та обслуговуванням клієнтів (наприклад, Salesforce, HubSpot, Zoho CRM);

– програмне забезпечення для управління відносинами з постачальниками (SRM): Спрямоване на управління відносинами з постачальниками, закупівлями та ланцюгом постачання (наприклад, SAP Ariba, Coupa, Oracle SCM);

– електронна комерція (eCommerce) та програмне забезпечення для управління продажами: Містить рішення для створення та управління онлайн-магазинами, обробки платежів, інвентаризації, лояльності клієнтів та іншого (наприклад, Shopify, WooCommerce, Magento).

2. Сфера застосування:

– медичне програмне забезпечення. Використовується в медичних установах для управління медичною інформацією, діагностики, лікування пацієнтів та іншого (наприклад, Epic, Cerner, Allscripts);

– освітнє програмне забезпечення. Охоплює навчальні платформи, системи управління навчанням (LMS), програми для дистанційного навчання та інше (наприклад, Moodle, Blackboard, Canvas);

– торгове програмне забезпечення. Використовується в роздрібній та оптовій торгівлі для автоматизації процесів продажу, інвентаризації, управління магазином та іншого (наприклад, POS-системи, Lightspeed Retail, Vend);

– виробниче програмне забезпечення. Використовується у виробництві для автоматизації виробничих процесів, планування

виробництва, контролю якості та іншого (наприклад, SAP ERP, Microsoft Dynamics, Plex Systems);

– програмне забезпечення для готельного бізнесу та ресторанів. Містить рішення для управління бронюваннями, обслуговуванням гостей, управлінням ресторанами та іншого (наприклад, Oracle Hospitality, Amadeus, OpenTable).

3. Функціональність та особливості:

– вебдодатки. Працюють у веббраузері та доступні через інтернет;

– мобільні додатки. Працюють на мобільних пристроях (смартфонах та планшетах) і доступні через мобільні платформи (iOS, Android);

– комп'ютерні додатки. Працюють на персональних комп'ютерах і доступні для завантаження та встановлення;

– хмарні додатки. Зберігають та оброблюють дані у хмарних сервісах і доступні з будь-якого пристрою за допомогою інтернету.

Кожна категорія може бути поділена на додаткові підкатегорії залежно від конкретних потреб та вимог користувачів.

1.4. Алгоритмічні мови й системи програмування

Алгоритмічні мови програмування (мови програмування для алгоритмів) – це мови програмування, які спеціалізуються на реалізації алгоритмів та обробці даних. Ці мови зазвичай надають розширені можливості для роботи зі структурами даних, операціями над ними. Найпопулярніші алгоритмічні мови програмування:

1. Python. Є однією з найпопулярніших мов програмування для реалізації алгоритмів. Вона має чистий синтаксис, широку підтримку структур даних та багато вбудованих бібліотек, що спрощують розробку та реалізацію різних алгоритмів.

2. Java. Є популярним вибором для програмування алгоритмів. Вона має вбудовану підтримку об'єктно-орієнтованого програмування, що полегшує створення складних алгоритмів та роботу зі структурами даних.

3. C++. C++ відома своєю швидкодією та можливостями маніпулювання пам’яттю, що робить її відмінним вибором для реалізації ефективних алгоритмів та оптимізації коду.

4. C. Мова програмування низького рівня, яка надає прямий доступ до апаратного забезпечення комп’ютера. Вона часто використовується для написання оптимізованих алгоритмів та роботи з пам’яттю.

5. R. Є мовою програмування та середовищем для статистичного аналізу й обробки даних. Вона надає багато вбудованих функцій і пакетів для статистичного аналізу, машинного навчання та візуалізації даних.

Існують також системи програмування, які спеціалізуються на розробці алгоритмів та обробці даних, такі як MATLAB, Octave, SciPy, NumPy та інші. Ці системи надають високорівневі інтерфейси та інструменти для реалізації різноманітних алгоритмів, особливо в галузі обробки сигналів, зображень, машинного навчання та наукових обчислень.

1.5. Рівні алгоритмічних мов програмування

Алгоритмічні мови програмування можна розділити на кілька рівнів відповідно до їхньої складності та призначення.

1. Високорівневі мови програмування. Ці мови дають змогу програмістам реалізовувати алгоритми та програми у вигляді, близькому до людської мови. Вони надають високий рівень абстракції, що робить їх більш зрозумілими та легкими для використання. Приклади: Python, Java, C#, JavaScript, Ruby.

2. Мови програмування низького рівня. Ці мови набагато ближчі до машинного коду та архітектури комп’ютера. Вони надають більше контролю над апаратним забезпеченням та оптимізацією програм, але можуть бути складними для розуміння. Приклади: C, C++, Асемблер.

3. Мови скриптів. Ці мови часто використовуються для автоматизації завдань, роботи зі сценаріями та швидкої розробки прототипів. Вони зазвичай інтерпретуються (а не компілюються)

та мають багато вбудованих функцій для роботи з файлами, текстом, мережами тощо. Приклади: Python, Perl, PHP, Bash.

4. Функціональні мови програмування. Ці мови спеціалізуються на функціональному програмуванні, де основними елементами програми є функції. Вони часто використовують рекурсію та функціональні конструкції для розв'язання різних проблем. Приклади: Haskell, Lisp, Scala, Erlang.

5. Мови програмування для паралельного програмування. Ці мови спеціалізуються на створенні паралельних програм, що виконуються на багатоядерних або розподілених системах. Вони надають спеціальні конструкції та бібліотеки для ефективного використання ресурсів і керування потоками виконання. Приклади: CUDA (для GPU), OpenMP, MPI.

Рівні алгоритмічних мов програмування відображаються на різних рівнях абстракції, контролю та складності, що робить їх підходящими для різних типів задач і вимог.

Мови програмування можна класифікувати з точки зору типізації даних. Типізація даних визначає, які типи даних можуть бути використані в програмі, а також правила перевірки їх відповідності та безпеки.

1. Статична типізація. У статично типізованих мовах програмування типи даних визначаються під час компіляції програми і перевіряються на відповідність під час компіляції. Приклади: Java, C++, C#, Kotlin.

2. Динамічна типізація. У динамічно типізованих мовах програмування типи даних визначаються під час виконання програми і перевіряються на відповідність під час виконання. Це надає більшу гнучкість, але може призводити до помилок під час виконання програми, якщо типи даних не відповідають очікуваним. Приклади: Python, JavaScript, Ruby.

3. Сильна (статична) типізація. У мовах із сильною типізацією типи даних строго визначені, і компілятор (або інтерпретатор) не дає можливості змішувати різні типи даних без використання явного приведення типів. Приклади: Java, C#, Haskell.

4. Слабка (динамічна) типізація. У мовах зі слабкою типізацією типи даних можуть автоматично перетворюватися один в

один без явного приведення типів, що може призводити до неочікуваних результатів. Приклади: JavaScript, PHP.

5. Нестрога (різновид слабкої) типізація. У нестрогих мовах типи даних можуть бути автоматично перетворені в інші типи даних без виведення помилки, але деякі операції можуть викликати помилки в час виконання. Приклади: Perl, PHP.

6. Статична інференція типів. У цьому підході типи даних виводяться автоматично компілятором на основі використання змінних і виразів у програмі. Це дає змогу уникнути необхідності в явній вказівці типів даних у деяких випадках, збільшуючи водночас зручність та безпеку коду. Приклади: Kotlin, TypeScript.

Вибір мови програмування залежить від конкретних потреб проєкту, особливостей виконання та вимог до типів даних. Кожен вид типізації має свої переваги та недоліки, і важливо обрати той, який найкраще відповідає конкретним вимогам проєкту та команди розробників.

Перелік питань для самоконтролю

1. Що таке програмне забезпечення?
2. Що таке програмне забезпечення системного рівня?
3. Що таке прикладне програмне забезпечення?
4. Які є рівні алгоритмічних мов програмування?
5. Назвіть принципи створення архітектури сучасних комп'ютерів?

Рекомендовані джерела:

Основні: 1; 2; 3; 4.

Допоміжні: 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Тема 2
Вступ до програмування мовою Python

2.1. Історія Python. Галузь застосування. Місце в сучасному світі.

2.2. Парадигми програмування.

2.3. Встановлення Python.

2.4. Компіляція, інтерпретація коду.

2.5. Інтерпретатор Python та його використання.

2.6. Середовища програмування мовою Python.

2.7. Динаміка та перспективи розвитку.

2.8. Дзен Python.

2.1. Історія Python. Галузь застосування. Місце в сучасному світі

Python представляє популярну високорівневу мову програмування, яка призначена для створення додатків різних типів. Зокрема, це і вебпрограми, і ігри, і робота з базами даних тощо. Досить велике поширення Python отримав у галузі машинного навчання, аналізу даних та досліджень штучного інтелекту.

Вперше мова Python була анонсована 1991 року голландським розробником Гвідо Ван Россумом. З того часу ця мова пройшла великий шлях розвитку. 2000 року вийшла версія 2.0, а 2008 року – версія 3.0. Актуальною версією на момент написання цього посібника є 3.12, яка вийшла в жовтні 2023 року.

Основні особливості мови програмування Python:

1. Скриптова мова.
2. Підтримка різних парадигм програмування, зокрема об’єктно-орієнтованої та функціональної парадигм.
3. Інтерпретація програм. Для роботи зі скриптами необхідний інтерпретатор, який запускає та виконує скрипт.
4. Портативність та платформонезалежність. Не має значення, яка у нас операційна система – Windows, Mac OS, Linux, нам достатньо написати скрипт, який запускатиметься на всіх цих ОС за наявності інтерпретатора.

5. Автоматичне керування пам’яті.

6. Динамічна типізація.

Скриптова мова (*scripting language*) – мова програмування, яка розроблена для запису так званих «сценаріїв», послідовностей операцій, які користувач може виконувати на комп’ютері. Сценарії зазвичай інтерпретуються, а не компілюються. У прикладній програмі, сценарій (скрипт) – це програма, яка автоматизує деяке завдання, яке без сценарію користувачу необхідно було б робити вручну. Мови таких скриптів спочатку орієнтувалися на використання як внутрішні керуючі мови у складних системах. Характерними особливостями таких мов є:

- їх інтерпретованість (компіляція або неможлива, або небажана);
- простий синтаксис;
- легка розширюваність.

Отже, вони ідеально підходять для використання в часто змінюваних програмах, дуже невеликих програмах або у випадках, коли для виконання операторів мови витрачається час, незрівнянно більший за час їх синтаксичного розбору інтерпретатором.

Скриптова мова використовується переважно в таких випадках.

1. Потрібно забезпечити програмовість без ризику дестабілізувати систему. Оскільки, на відміну від плагінів, скрипти інтерпретуються, а не компілюються, неправильно написаний скрипт виведе діагностичне повідомлення, а не призведе систему до краху.

2. Важливий виразний код. По-перше, чим складніша система, тим більше коду доводиться писати «тому, що це потрібно». По-друге, у скриптовій мові може бути зовсім інша концепція програмування, ніж в основній програмі. По-третє, скриптова мова має власний проблемно-орієнтований набір команд, і один рядок скрипту може робити те саме, що кілька десятків рядків мовою нижчого рівня.

3. Якщо потрібна платформонезалежність.

З іншого боку, оскільки скрипти інтерпретуються з початкового коду динамічно при кожному виконанні, вони виконуються зазвичай значно повільніше за готові програми. Тому скриптові мови, зокрема Python, не застосовуються для написання програм,

що потребують оптимальності й швидкості виконання. Але через простоту вони часто застосовуються для написання невеликих, одноразових програм, прототипів.

Скриптові мови можна розділити на мови динамічного розбору і передкомпільовані. Мови першого типу читають інструкції з файлу програми мінімально потрібними блоками і виконують ці блоки, не читаючи подальший код. Передкомпільовані мови – читають всю програму, компілюють її в машинний код або байт-код (або в якийсь внутрішній формат) і лише потім виконують отриманий код.

Виконання програми на Python виглядає так. Спочатку ми пишемо в текстовому редакторі скрипт з набором виразів цією мовою програмування. Передаємо цей скрипт виконання інтерпретатору. Інтерпретатор транслює код у проміжний байткод, а потім віртуальна машина переводить отриманий байткод на набір інструкцій, що виконуються операційною системою.

Варто зазначити, що хоча формально трансляція інтерпретатором вихідного коду в байткод і переведення байткоду віртуальною машиною в набір машинних команд представляють два різні процеси, але фактично вони об’єднані в інтерпретаторі.

Python – дуже проста мова програмування, вона має короткий і водночас досить простий і зрозумілий синтаксис. Відповідно його легко вивчати, і власне це одна з причин, через яку він є однією з найпопулярніших мов програмування саме для навчання. Python також популярний не лише у сфері навчання, а й у написанні конкретних програм, зокрема комерційного характеру. Немалою мірою тому для цієї мови написано багато бібліотек, які ми можемо використовувати. Крім того, у цієї мови програмування дуже велике співтовариство програмістів, в інтернеті можна знайти з цієї мови безліч корисних матеріалів, прикладів, отримати кваліфіковану допомогу фахівців.

Пакети та бібліотеки. Інтерпретатор Python супроводжується достатнім функціоналом, який дає змогу створювати програми цією мовою. Проте цього функціоналу може виявитися замало для ряду завдань. Але через велику спільноту розробників цією мовою по всьому світу також є велика екосистема різних пакетів і

бібліотек, які можна використовувати для різних цілей. Наприклад, для створення графічних програм (Tkinter, PyQt / PySide, wxPython, DearPyGui, EasyGUI), для створення мобільних додатків (Kivy, Toga), для створення вебзастосунків (Django, Flask, FastAPI, Pylons, Bottle, CherryPy, TurboGears, Nagare), для автоматизації процесів (Selenium, robotframework, pywinauto, Lettuce, Behave, Requests), для роботи з різними типами файлів (OpenPyXL (Excel), lxml (XML), ReportLab / borb (PDF), pdfrw / PyPDF2 (PDF), Pandas (CSV та Excel), для машинного навчання, штучного інтелекту, Data Science (Pandas, SciPy, PyTorch, Matplotlib, Theano, Tensorflow, OpenCV, Scikit-Learn, Keras, NumPy), для візуалізації (Matplotlib, Seaborn, Plotly, Bokeh, Altair, HoloViews).

2.2. Парадигми програмування

Парадигми програмування – це основні концепції та підходи до розробки програмного забезпечення, які визначають стиль написання коду й способи організації програм.

1. Імперативне програмування. Програміст описує кроки, які необхідно виконати для отримання потрібного результату. Це може бути досягнуто за допомогою інструкцій присвоєння значень, циклів, умовних операторів та інших засобів управління потоком виконання програми.

2. Декларативне програмування. Програміст описує, що потрібно зробити, а не як це робити. Описується бажаний результат, а система вирішує, як його досягти. Прикладами декларативного програмування є функціональне та логічне програмування.

3. Функціональне програмування. Це парадигма[1], де програма розглядається як набір функцій, які обробляють дані. Основні принципи цієї парадигми містять уникнення стану та змінних, використання чистих функцій, які не мають побічних ефектів, та використання рекурсії.

[1] Парадигма програмування – це система ідей і понять, які визначають стиль написання комп'ютерних програм, а також спосіб мислення програміста.

4. Об’єктно-орієнтоване програмування (ООП). У цій парадигмі програмування програма розглядається як набір об’єктів, які взаємодіють один з одним. Кожен об’єкт має свої властивості (атрибути) та може виконувати дії (методи). ООП використовує концепції спадкування, поліморфізму та інкапсуляції.

5. Логічне програмування. Програми описуються як набір правил та фактів, з якими система робить логічні виводи. Основний інструмент цієї парадигми – логічні програмувальні мови, такі як Prolog.

6. Імперативно-функціональне програмування. Комбінація імперативної та функціональної парадигм програмування, де використовуються як імперативні, так і функціональні конструкції для досягнення мети.

Ці парадигми не є взаємовиключними, і часто програми можуть використовувати різні підходи залежно від ситуації та потреб проєкту. Багато мов програмування підтримують кілька парадигм програмування і дають змогу програмістам вибирати той підхід, який найбільше підходить для конкретної задачі.

2.3. Встановлення Python

Щоб встановити Python на ваш комп’ютер, Вам потрібно виконати такі кроки:

1. Завантажте Python: Перш за все, Вам потрібно завантажити виконуваний файл для встановлення Python з [офіційного вебсайту] (https://www.python.org/downloads/). Оберіть версію Python, яку ви хочете встановити. Рекомендовано завантажити останню стабільну версію, яка доступна.

2. Запустіть виконуваний файл для встановлення: Після завантаження виконуваного файлу запустіть його та слідуйте інструкціям майстра встановлення. Вам можуть бути запропоновані різні параметри, такі як шлях для встановлення та налаштування, які ви можете змінити за вашими уподобаннями.

3. Перевірте правильність встановлення: Після завершення встановлення відкрийте термінал або командний рядок і введіть

команду `python`. Якщо Python успішно встановлено, ви повинні побачити повідомлення про версію Python, яка відображатиметься у вікні терміналу. Наприклад:

```
Python 3.9.5 (default, May 27 2021, 13:30:53)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

4. Налаштуйте змінні середовища (опціонально). Щоб мати змогу запускати Python з будь-якого місця у вашій системі, вам може бути потрібно додати шлях до встановленого Python до вашої змінної середовища.

Після виконання цих кроків ви зможете почати використовувати Python для розробки програмного забезпечення, виконання скриптів та багато іншого.

2.4. Компіляція, інтерпретація коду

Компіляція та інтерпретація – це два основних підходи до виконання програмного коду.

1. Компіляція. У компіляції програмний код перетворюється з вихідного коду в машинний код або код байтів (для виконання на віртуальній машині). Компіляція виконується за допомогою програми, яка називається компілятором. Компілятор аналізує вихідний код програми та генерує еквівалентний машинний код або код байтів. Отриманий машинний код може бути виконаний на цільовій платформі без необхідності повторного перекладу. Приклади: C, C++, Java (з деякими винятками), Go.

2. Інтерпретація. Під час інтерпретації програмний код виконується рядок за рядком у режимі реального часу. Інтерпретатор виконує вихідний код програми, читаючи та виконуючи його послідовно. Інтерпретація зазвичай пов’язана з мовами програмування високого рівня, що забезпечують динамічну типізацію та динамічну пам’ять, а також з мовами скриптів. Приклади: Python, Ruby, JavaScript, PHP.

Обидва підходи мають свої переваги та недоліки. Компіляція зазвичай забезпечує кращу продуктивність виконання, оскільки машинний код може бути оптимізований під конкретну платформу, але вона може вимагати часу на перетворення вихідного коду. Інтерпретація забезпечує більшу гнучкість та простоту виконання, оскільки ви можете відразу виконувати зміни в коді без необхідності повторної компіляції, але може бути повільніше через переклад коду в реальному часі.

2.5. Інтерпретатор Python та його використання

Інтерпретатор Python – це програмне забезпечення, яке забезпечує виконання програм, написаних мовою Python. Це основний інструмент для розробки та виконання Python-коду.

Ключові аспекти використання інтерпретатора Python:

1. Виконання скриптів та програм. Інтерпретатор Python виконує програми та скрипти, написані мовою програмування Python. Ви можете виконувати Python-код з командного рядка або інтерактивно у відомому як «Python Shell» або «Python REPL».

2. Розробка програмного забезпечення. Python широко використовується для розробки різноманітних програм, охоплюючи вебдодатки, мобільні додатки, наукові обчислення, штучний інтелект, машинне навчання, ігри та багато іншого. Інтерпретатор Python дає змогу програмістам створювати, тестувати та відлагоджувати свої програми швидко й ефективно.

3. Інтерактивна робота. Інтерпретатор також може використовуватися для виконання коду в режимі інтерактивного введення. Ви можете вводити та виконувати Python-код по одній команді або фрагменту коду одразу, отримуючи миттєві результати.

4. Експерименти та прототипування. Python є популярним вибором для швидких експериментів та прототипування через свою зручність та виразність синтаксису. Інтерпретатор дає змогу швидко тестувати різні ідеї та алгоритми без необхідності компіляції коду.

5. Навчання програмування. Python дуже популярний серед початківців у програмуванні через свій простий та зрозумілий синтаксис.

Інтерпретатор Python доступний для багатьох операційних систем (у Windows, macOS, різні дистрибутиви Linux).

2.6. Середовища програмування мовою Python

Існує декілька середовищ програмування для мови Python, які спрощують розробку, тестування та відлагодження коду.

1. PyCharm. Є одним з найпопулярніших середовищ програмування для Python. Він надає багато функцій, таких як автодоповнення коду, інтеграція з системами контролю версій, відладка, підтримка віртуальних середовищ, тестування та багато іншого.

2. Visual Studio Code (VS Code). Безкоштовний та легкий редактор коду, який надає широкий спектр розширень для підтримки роботи з Python, охоплюючи автодоповнення, відладку, інтеграцію з Git та інші корисні функції.

3. Jupyter Notebook. Вебсередовище, яке дає змогу виконувати Python-код у вигляді окремих блокнотів (як документів), що містять код, текст, графіки та інші елементи. Він часто використовується для наукових обчислень та аналізу даних.

4. Spyder. Середовище програмування, спеціалізоване на наукових обчисленнях та аналізі даних. Воно має вбудовані інструменти для відлагодження, візуалізації даних та інші корисні функції.

5. Atom. Ввільний редактор коду, який має велику кількість розширень для підтримки роботи з Python. Він надає зручний інтерфейс, можливості кастомізації та інші корисні функції.

6. Sublime Text. Швидкий та легкий редактор коду, який підтримує роботу з Python через різноманітні розширення та плагіни.

Усі ці та інші середовища програмування мають різні переваги й недоліки, і вибір залежить від ваших потреб та особистих уподобань. Більшість з них доступні як безкоштовне програмне забезпечення та можуть бути легко встановлені на ваш комп’ютер.

2.7. Динаміка та перспективи розвитку

Python за останні роки пережив значний ріст популярності та використання в різних галузях та проєктах. Деякі ключові аспекти динаміки та перспектив розвитку Python:

1. Зростання популярності. Python став однією з найпопулярніших мов програмування у світі за останні кілька років. Це підтверджують різноманітні рейтинги мов програмування, такі як TIOBE та Stack Overflow Developer Survey.

2. Застосування в широкому спектрі галузей. Python використовується в різних галузях, охоплюючи веб-розробку, наукові обчислення, обробку даних, штучний інтелект, машинне навчання, інтернет-речі та багато іншого. Ця універсальність робить його привабливим для різноманітних проєктів і компаній.

3. Розвиток екосистеми. Екосистема Python постійно розвивається, з'являються нові бібліотеки, фреймворки та інструменти, які сприяють швидкому та ефективному розробленню програмного забезпечення. Такі бібліотеки, як NumPy, Pandas, TensorFlow, PyTorch, Django, Flask та інші, стали невід'ємною частиною екосистеми Python.

4. Підтримка спільноти. Python має велику та активну спільноту розробників, яка активно співпрацює, розробляє та підтримує цю мову програмування. Це сприяє швидкому розвитку мови та широкому розповсюдженню знань і практик серед програмістів.

5. Постійне вдосконалення. Розробники Python постійно працюють над вдосконаленням мови та забезпеченням її актуальністю й конкурентоспроможністю в сучасному програмуванні. Кожний новий випуск мови (наприклад, Python 3.12) містить нові функції, покращення та оптимізації.

6. Використання в навчанні. Python став популярним вибором для навчання програмування у школах, університетах та на онлайн-курсах. Його зрозумілий синтаксис та велика кількість ресурсів для самостійного навчання роблять його привабливим для початківців у програмуванні.

Загалом, Python залишається однією з найбільш впливових та широко використовуваних мов програмування, і його майбутнє виглядає яскраво, оскільки він продовжує розвиватися та пристосовуватися до змін у вимогах і технологіях.

2.8. Дзен Python

«Дзен Python» – це набір принципів та філософія, які описують кращі практики та підходи до написання коду на мові програмування Python. Цей термін походить від ідеї «зен» у традиційній японській культурі, яка спрямована на досягнення гармонії та спокою через природу, медитацію й саморозвиток.

Декілька ключових принципів «Дзен Python».

1. Читабельність коду. Python прагне до того, щоб код був легким для читання й розуміння. Це означає, що він має бути написаний так, щоб його було легко розгорнути та зрозуміти іншим програмістам.

2. Експресивність і простота. Python надає зручні та експресивні засоби для вирішення складних завдань. Він прагне до того, щоб програми були написані просто й зрозуміло, уникайте надмірного складного коду.

3. Використання зручних структур даних та ідіом Python. Python має вбудовані структури даних та ідіоми, які сприяють написанню ефективного й зрозумілого коду. Це охоплює використання списків, словників, генераторів, спискових виразів та інших вбудованих засобів мови.

4. Використання виразності Python. Python ставить на перше місце виразність коду. Це означає, що код повинен бути написаний так, щоб його було легко читати та зрозуміти, навіть для тих, хто не має глибоких знань мови.

5. Простота й зручність у вирішенні проблем. Python прагне до того, щоб рішення проблем були простими та зручними для використання. Це охоплює використання інтуїтивно зрозумілих методів та функцій для розв’язання завдань.

Загалом «Дзен Python» наголошує на важливості простоти, читабельності та виразності коду, які є важливими принципами для написання ефективного й зрозумілого програмного забезпечення.

Перелік питань для самоконтролю

1. Хто є розробником мови Python?
2. Звідки походить назва мови програмування Python?
3. Для яких цілей використовується Python?
4. Як перейти в режим інтерактивного інтерпретатора?
5. Яке розширення мають файли із програмами, написаними на мові Python?
6. Як запустити програму, що міститься у файлі, на виконання у термінальному вікні?

Рекомендовані джерела:

Основні: 1; 2; 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Тема 3
Змінні в мові Python

3.1. Іменування змінних, список ключових слів та інструкція для їх виведення.

3.2. Базові типи даних скриптової мови програмування Python.

3.3. Способи присвоювання значення змінним.

3.4. Визначення та перевірка типу даних.

3.5. Способи перетворення типів даних.

3.6. Видалення змінної.

3.7. Анотація змінних.

3.8. Консольне введення та виведення даних.

3.1. Іменування змінних, список ключових слів та інструкція для їх виведення

Змінні призначені для зберігання даних. Назва змінної в Python повинна починатися з алфавітного символу або символу підкреслення і може містити алфавітно-цифрові символи і знак підкреслення. Крім того, назва змінної не повинна співпадати з назвою ключових слів мови Python. До таких слів, зокрема, належать: *False, await, else, import, pass, None, break, except, in, raise, True, class, finally, is, return, and, continue, for, lambda, try, as, def, from, nonlocal, while, assert, del, global, not, with, async, elif, if, or, yield.*

Наприклад, створимо змінну: *name = "Tom"*. Тут визначено змінну *name,* яка зберігає рядок (текст) *"Tom"*.

У Python застосовується два типи найменування змінних: *camel case* та *underscore notation.*

Camel case має на увазі, що кожне нове підслово в найменуванні змінної починається з великої літери. Наприклад:

```
userName = "Tom"
```

Underscore notation передбачає, що слова в найменуванні змінної поділяються знаком підкреслення. Наприклад:

```
user_name = "Tom"
```

Також треба враховувати регістрозалежність, тому змінні *name* і *Name* будуть представляти різні об'єкти.

```
# Дві різні змінні
name = "Tom"
Name = "Tom"
```

Визначивши змінну, ми можемо її використовувати в програмі. Наприклад, спробувати вивести її вміст на консоль за допомогою вбудованої функції *print:*

```
name = "Tom"  # Визначення змінної name
print(name)    # Виведення значення змінної name на консоль
```

Відмінною рисою змінної є те, що ми можемо змінювати її значення протягом роботи програми.

```
name = "Tom"    # змінна name дорівнює "Tom"
print(name)   # виводить: Tom
name = "Bob"   # змінюємо значення на "Bob"
print(name),# виводить: Bob
```

Програма Python складається з набору інструкцій. Кожна інструкція міститься на новому рядку. Наприклад:

```
print(2 + 3)
print("Hello")
```

Велику роль у Python відіграють відступи. Неправильно поставлений відступ є помилкою. Наприклад, у наступному випадку ми отримаємо помилку, хоча код буде практично аналогічний наведеному вище:

```
print(2 + 3)
   print("Hello")
```

Тому варто поміщати нові вказівки спочатку рядка. У цьому одна з важливих відмінностей Python від інших мов програмування, наприклад таких як C # або Java.

Однак варто враховувати, що деякі конструкції мови можуть складатися з декількох рядків. Наприклад, умовна конструкція if:

```
if 1 < 2:
    print("Hello")
```

У разі якщо 1 менше 2, то виводиться рядок *" Hello "*. І тут має бути відступ, оскільки інструкція *print("Hello")* використовується не як така, а як частина умовної конструкції if. Причому відступ, згідно з посібником з оформлення коду, бажано робити з такої кількості прогалин, яка кратна 4 (тобто 4, 8, 16 тощо) Таких конструкцій не так багато, тому особливої плутанини щодо того де треба, а де не треба ставити прогалини, не повинно виникнути.

Регістрозалежність. Python – регістрозалежна мова, тому вирази *print* і *Print* або *PRINT* представляють різні вирази. І якщо замість методу *print* для виведення на консоль ми спробуємо використати метод *Print*:

```
Print("Hello World")
```

то нічого не вийде.

Коментарі. Для позначки (пояснення), що робить та чи інша ділянка коду, застосовуються коментарі. Під час трансляції та виконання програми інтерпретатор ігнорує коментарі, тому вони не впливають на роботу програми. Коментарі в Python бувають блокові та малі.

Рядкові коментарі передуються знаком решітки – #. Вони можуть розташовуватися на окремому рядку:

```
# Виведення на консоль
# повідомлення Hello World
print("Hello World")
```

Будь-який набір символів після знак # представляє коментар. Тобто у прикладі вище перші два рядки коду є коментарями.

Також вони можуть розташовуватися на тому самому рядку, що й інструкції мови після інструкцій, що виконуються:

```
print("Hello World") # Виведення повідомлення на консоль
```

У блокових коментарях до і після тексту коментаря ставляться три одинарні лапки. Наприклад:

```
'''
Виведення на консоль
повідомлення Hello World
'''
print("Hello World")
```

Основні функції. Python надає ряд вбудованих функцій. Деякі з них використовуються дуже часто, особливо в початкових етапах вивчення мови.

Основною функцією виведення інформації на консоль є функція *print()*. Як аргумент у цю функцію передається рядок, який ми хочемо вивести:

```
print("Hello Python")
```

Якщо нам необхідно вивести кілька значень на консоль, ми можемо передати їх у функцію *print*() через кому:

```
print("Full name:", "Tom", "Smith")
```

У результаті всі передані значення «склеяться» через прогалини в один рядок.

Функція *input()* відповідає за введення інформації. Як необов'язковий параметр ця функція приймає запрошення до введення та повертає введений рядок, який ми можемо зберегти в змінний:

```
name = input("Введіть ім'я: ")
print("Привіт", name)
```

3.2. Базові типи даних скриптової мови програмування Python

Змінна зберігає дані одного з типів даних. У Python існує декілька різних типів даних. Розглянемо лише базові типи: *bool, int, float, complex і str*.

Логічні значення. Тип *bool* представляє два логічні значення: *True* (вірно, істина) або *False* (невірно, брехня). Значення *True* полягає в тому, щоб показати, що щось істинно. Тоді як значення *False*, навпаки, показує, що щось хибне. Приклад змінних цього типу:

```
isMarried = False
print(isMarried)    # False
isAlive = True
print(isAlive)     # True
```

Цілі числа. Тип *int* є цілим числом, наприклад, 1, 4, 8, 50. Приклад:

```
age = 21
print("Вік:", age)     # Вік: 21
count = 15
print("Кількість:", count)    # Кількість: 15
```

За замовчуванням стандартні числа розцінюються як числа в десятковій системі. Але Python також підтримує числа у двійковій, вісімковій та шістнадцятковій системах.

Для вказівки, що число представлене у двійковій системі, перед числом ставиться префікс 0b:

```
a = 0b11
b = 0b1011
c = 0b100001
print(a)    # 3 у десятковій системі
print(b)    # 11 у десятковій системі
print(c)    # 33 у десятковій системі
```

Для вказівки, що число представлене вісімковою системою, перед числом ставиться префікс 0o:

```
a = 0o7
b = 0o11
c = 0o17
print(a)    # 7 у десятковій системі
print(b)   # 9 у десятковій системі
print(c)   # 15 у десятковій системі
```

Для вказівки, що число представлене шістнадцятковою системою, перед числом ставиться префікс 0x:

```
a = 0x0A
b = 0xFF
c = 0xA1
print(a)     # 10 у десятковій системі
print(b)    # 255 у десятковій системі
print(c)    # 161 у десятковій системі
```

У будь-якій системі ми не передали число в функцію *print* для виведення на консоль, воно за умовчанням буде виводитися в десятковій системі.

Дробові числа. Тип *float* є число з плавальною точкою, наприклад, 1.2 або 34.76. Як роздільник цілої та дробової частин використовується крапка.

```
height = 1.68
pi = 3.14
weight = 68.
print(height)    # 1.68
print(pi)         # 3.14
print(weight)    # 68.0
```

Число з плавальною точкою можна представити в експоненційному записі:

```
X =  3.9e3
print(x)   # 3900.0
x =  3.9e-3
print(x)   # 0.0039
```

Число типу *float* може мати лише 18 значущих символів. Так, у цьому випадку використовуються лише два символи – 3.9. І якщо число занадто велике чи занадто мале, ми можемо записувати число в подібній нотації, використовуючи експоненту. Число після експоненти вказує ступінь числа 10, на яке треба помножити основне число – 3.9.

Комплексні числа. Тип complex представляє комплексні числа у форматі дійсна_частина + уявна_частина – після уявної частини вказується суфікс *j*

```
complexNumber = 1+2j
print(complexNumber)  # (1+2j)
```

Рядки. Тип *str* представляє рядки (звичайний текст). Рядок представляє послідовність символів, укладену в одинарні або подвійні лапки, наприклад *'hello'* та *"hello"*. У Python 3.x рядки представляють набір символів у кодуванні *Unicode*

```
message = "Hello World!"
print(message)  # Hello World!
name = 'Tom'
print(name)  # Tom
```

Водночас якщо рядок має багато символів, його можна розбити на частини і ці частини розмістити на різних рядках коду. У цьому випадку весь рядок береться в круглі дужки, а її окремі частини – у лапки:

```
text = ("Laudate omnes gentes laudate "
      "Magnificat in secula ")
print(text)
```

Якщо ми хочемо визначити багаторядковий текст, такий текст розміщують у потрійні подвійні чи потрійні одинарні лапки:

```
'''
Це коментар
'''
text = '''Laudate omnes gentes laudate
Magnificat in secula
Et anima mea laudate
Magnificat in secula
'''
print(text)
```

Під час використання потрійних одинарних лапок не варто плутати їх із коментарями: якщо текст у потрійних одинарних лапках присвоюється змінній, то це рядок, а не коментар.

3.3. Способи присвоювання значення змінним

У Python є кілька способів присвоєння значень змінним:

1. Присвоєння одного значення змінній. Ви можете присвоїти одне значення одній змінній, використовуючи оператор присвоєння `=`:

```
x = 10
```

2. Присвоєння одного значення декільком змінним. Ви можете присвоїти одне значення декільком змінним одночасно:

```
x = y = z = 10
```

3. Присвоєння декількох значень різним змінним. Ви можете присвоїти декілька значень декільком змінним одночасно:

```
x, y, z = 10, 20, 30
```

4. Масивне присвоєння. Ви можете присвоїти значення декільком змінним, використовуючи масив (або кортеж) значень:

```
values = [10, 20, 30]
x, y, z = values
```

5. Присвоєння з використанням розпакування. Ви можете присвоїти значення одночасно з використанням розпакування:

```
x, y = 10, 20
```

6. Присвоєння з використанням методів та функцій. Ви можете присвоїти значення змінній, використовуючи результат виклику методу або функції:

```
x = len("Hello")
```

Ці способи дають змогу зручно та ефективно присвоювати значення змінним у Python залежно від вашої потреби.

Моржове присвоєння – це нова функціональність, яка була впроваджена у Python 3.8. Це оператор присвоєння, який використовує символ ":=" і називається оператором "моржа" через свою схожість з формою моржа. Приклад:

```
# Зазвичай ви могли записати таке присвоєння
x = 10
# За допомогою моржового присвоєння ви можете прочитати значення і зразу його вивести
if (x := 10) == 10:
    print("x дорівнює 10")
```

Основна ідея полягає в тому, що ви можете присвоїти значення змінній та одночасно використовувати це значення для порівняння, виклику функцій чи будь-якої іншої операції. Це зручно, коли ви хочете використовувати те саме значення двічі без додаткового рядка коду.

Це особливо корисно у виразах if, while та в умовних виразах, але може використовуватися в будь-якому місці, де дозволено присвоєння. Однак потрібно бути обережним, оскільки використання цього оператора може призвести до менш зрозумілого коду, якщо він занадто складний або в разі зловживання цього оператора.

3.4. Визначення та перевірка типу даних

Python є мовою з динамічною типізацією. Це означає, що змінна не прив'язана до певного типу. Тип змінної визначається виходячи зі значення, яке їй надано. Так, під час присвоєння рядка в подвійних чи одинарних лапках змінна має тип *str*. Під час присвоєння цілого числа Python автоматично визначає тип змінної як *int*. Щоб визначити змінну як об'єкт *float*, їй присвоюється дробове число, в якому роздільником цілої та дробової частини є крапка.

Водночас у процесі роботи програми ми можемо змінити тип змінної, надавши їй значення іншого типу:

```
userId = "abc"  # тип str
print(userId)
userId = 234  # тип int
print(userId)
```

За допомогою вбудованої функції *type()* можна дізнатися про поточний тип змінної:

```
userId = "abc"     # тип str
print(type(userId)) # <class 'str'>
userId = 234       # тип int
print(type(userId)) # <class 'int'>
```

3.5. Способи перетворення типів даних

У Python існують кілька способів перетворення типів даних, що надають гнучкість та зручність для роботи з даними.

1. Вбудовані функції. Python має кілька вбудованих функцій для перетворення типів даних. Найпоширеніші з них:

- *int():* перетворює об'єкт у ціле число;
- *float():* перетворює об'єкт у число з рухомою комою;
- *str(`:* перетворює об'єкт у рядок;
- *bool():* перетворює об'єкт у булеве значення.

Наприклад:

```
x = "10"
y = int(x)  # y буде 10, ціле число
```

2. Методи об'єктів. Деякі об'єкти мають методи для перетворення свого типу даних. Наприклад, методи *list()* та *tuple()* можуть бути використані для перетворення інших ітерабельних об'єктів у списки та кортежі відповідно.

3. Оператори. Оператори такі як `+`, `-`, `*`, `/` можуть автоматично виконувати приведення типів, якщо це необхідно. Наприклад:

```
x = 10
y = 3.5
z = x + y  # z буде 13.5, число з рухомою комою
```

4. Функції форматування рядків. Функції форматування рядків, такі як *format()* або f-рядки, також можуть використовуватися для автоматичного перетворення типів даних при вставці значень у рядок.

Наприклад:

```
x = 10
y = "Value: {}".format(x)  # y буде "Value: 10", рядок
```

5. JSON серіалізація. Модуль *json* в Python дає змогу перетворювати дані між Python об'єктами та рядками JSON, що дає можливість легко перетворювати типи даних для збереження або їх передачі.

Наприклад:

```
import json
data = {"name": "John", "age": 30}
json_string = json.dumps(data)  # Серіалізація в JSON рядок
```

3.6. Видалення змінної

У Python ви можете видалити змінну за допомогою оператора *del*. Видалення змінної призводить до того, що посилання на об'єкт, яке було пов'язане з цією змінною, видаляється, і пам'ять, яку займав цей об'єкт, стає доступною для збору сміття. Приклад видалення змінної:

```
x = 10
print(x)  # Виведе: 10
del x  # Видалення змінної x
```

Після видалення змінна *x* буде недоступною:

```
print(x)  # Викличе помилку: NameError: name 'x' is not defined
```

Варто зазначити, що видалення змінної не означає видалення об'єкта, на який вона вказує. Якщо інші змінні або об'єкти використовують той самий об'єкт, то він продовжить існувати, поки на нього є посилання. Видалення змінної лише знищує посилання на об'єкт.

Також варто пам'ятати, що видалення змінної може бути корисним, коли ви більше не потребуєте використовувати цю змінну, особливо якщо вона велика та Вам потрібно звільнити пам'ять.

3.7. Анотація змінних

Анотація змінних – це процес додавання метаданих[2] до змінних у програмі для покращення зрозуміння їх типів та призначення. Ця практика, яка поширюється в деяких мовах програмування, дає змогу розробникам та іншим членам команди отримувати більше інформації про те, які дані повинні зберігатися в конкретних змінних. Деякі основні аспекти анотації змінних.

1. Тип даних. Анотація змінних дає змогу вказати, який тип даних повинен зберігатися у змінній. Це допомагає забезпечити правильність та безпеку під час використання цієї змінної в програмі.

2. Документація. Анотація змінних також може містити коментарі або документацію, яка пояснює призначення цієї змінної, її значення та як вона повинна використовуватися.

3. Контроль типів. У деяких мовах програмування, де підтримується статична типізація, анотація змінних може допомогти компілятору перевірити, чи відповідає тип даних, вказаний у анотації, типу даних, встановленому в коді.

4. Зручність налагодження. Анотація змінних може також полегшити налагодження програми, оскільки вона надає додаткову інформацію про дані, які використовуються в різних частинах програми.

Наприклад, у мові Python анотація змінних може бути виконана за допомогою спеціальних коментарів або анотацій функцій. Наприклад:

```
# Анотація змінної age як ціле число
age: int = 25

# Анотація змінної name як рядок
name: str = "John Doe"
```

[2] Метада́ні (*metadata*, походить від лат. *Meta* — мета, кінцевий пункт, межа, край і, власне, дані), у загальному випадку, — це дані, що характеризують або пояснюють інші дані. Наприклад, значення «123456» само по собі недостатньо виразне. А якщо значення «123456» описати як «поштовий індекс», то в цьому контексті значення «123456» стає більш осмислене. Метадані — це дані про дані.. Цей термін у широкому сенсі слова вживають щодо будь-яких «даних про дані»: іменах таблиць, колонок у таблиці, програм і тому подібне.

В інших мовах програмування, таких як Java або TypeScript, анотації змінних можуть бути виконані за допомогою спеціальних ключових слів у оголошенні змінної. Наприклад:

```
// Анотація змінної age як ціле число
int age = 25;
// Анотація змінної name як рядок
String name = "John Doe";
```

У Python можна використовувати анотації для анотування даних, функцій та класів. Це дає змогу додавати додаткову інформацію до коду, таку як типи даних аргументів та значень, а також документацію. Хоча анотації є необов'язковими для виконання коду Python, вони можуть бути корисними для документування та зрозуміння коду, особливо в більших проєктах.

Приклади використання анотацій для анотування даних, функцій та класів у Python:

1. Анотація змінних:

```
age: int = 25
name: str = "John Doe"
```

2. Анотація функцій:

```
def greet(name: str) -> None:
    print(f"Hello, {name}!")
def add(x: int, y: int) -> int:
    return x + y
```

3. Анотація класів:

```
class Person:
    name: str
    age: int
    def __init__(self, name: str, age: int) -> None:
        self.name = name
        self.age = age
    def greet(self) -> None:
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

У цих прикладах анотації вказують типи даних або значень, які повинні бути передані або повернуті функціями та методами.

Важливо зауважити, що анотації в Python є лише метаданими і не впливають на фактичне виконання програми. Вони не перевіряються інтерпретатором Python за замовчуванням, але можуть бути використані сторонніми інструментами або бібліотеками для аналізу коду або для автоматичної генерації документації.

У Python існують декілька бібліотек, які можна використовувати для анотації даних. Ці бібліотеки дають змогу вказати типи даних, атрибути та метадані для класів, функцій, методів та інших об'єктів. Деякі з найпопулярніших бібліотек для анотації даних у Python:

1. *typing* – це вбудована бібліотека в Python, яка надає набір стандартних типів даних та інструменти для анотації коду. За допомогою *typing* можна вказати типи даних, створити аліаси типів (псевдонім типу), визначити опціональні аргументи та повернуті значення, а також створювати складні типи даних, такі як кортежі, списки, словники тощо. Приклад:

```
from typing import List, Tuple

def process_data(data: List[Tuple[str, int]]) -> None:
    # функція приймає список кортежів, кожен з яких має перший елемент рядка і другий – ціле число
    pass
```

2. *pydantic* – це бібліотека для валідації даних та створення об'єктів з анотаціями. Вона дає змогу визначати моделі даних з анотаціями типів, обов'язкових та необов'язкових полів, валідаційних правил тощо. Приклад:

```
from pydantic import BaseModel

class User(BaseModel):
    name: str
    age: int

user = User(name="John", age=30)
```

3. *attrs* – це бібліотека для створення класів з анотаціями атрибутів та метаданих. Вона дає змогу створювати прості та імутабельні об’єкти (об’єкти, які після їх створення змінити не можна) з анотаціями типів даних. Приклад:

```
import attr

@attr.s(auto_attribs=True)
class Person:
   name: str
   age: int

person = Person(name="John", age=30)
```

Ці бібліотеки допомагають зробити код більш зрозумілим, безпечнішим та ефективнішим через додавання анотацій даних. Вони забезпечують можливість валідації та визначення типів даних, що полегшує розробку та підтримку програм.

3.8. Консольне введення та виведення даних

У Python для консольного введення та виведення даних використовуються функції *input()* та *print()*.

1. Консольне введення даних:

```
name = input("Введіть ваше ім'я: ")
age = int(input("Введіть ваш вік: "))
print(f"Привіт, {name}! Ваш вік - {age} років.")
```

У цьому прикладі функція *input()* використовується для зчитування введення користувача з консолі. За замовчуванням, *input()* повертає рядок, тому якщо потрібно прочитати ціле число або інший тип даних, варто скористатися відповідними функціями перетворення типів, наприклад, *int()* для перетворення на ціле число.

2. Консольне виведення даних:

```
name = "John"
age = 30
print("Ім'я:", name)
print("Вік:", age)
```

Функція *print()* використовується для виведення даних на консоль. Вона може приймати будь-яку кількість аргументів та автоматично перетворює їх у рядки для виведення. Можна також використовувати форматовані рядки або f-рядки для виведення даних у зручному форматі.

Ці функції дають змогу взаємодіяти з користувачем через консоль, отримуючи введення та виводячи результати обробки.

У функцію *input()* також може надсилатись запрошення до введення даних. А результат введення ми можемо зберегти у змінну. Наприклад, визначимо код для введення користувачем імені:

```
name = input("Введіть своє ім'я: ")
print(f"Ваше ім'я: {name}")
```

У цьому випадку функцією *input()* передається запрошення до введення у вигляді рядка "Введіть своє ім’я: ". Результат функції – результат введення користувача передається змінною *name*. Потім ми можемо вивести значення цієї змінної консоль за допомогою функції *print()*. Приклад роботи коду:

```
Введіть своє ім'я: Eugene
Ваше ім'я: Eugene
```

Ще приклад із введенням кількох значень:

```
name = input("Your name: ")
age = input("Your age: ")
print(f"Name: {name} Age: {age}")
```

Приклад роботи програми:

```
Your name: Tom
Your age: 37
Name: Tom Age: 37
```

Варто враховувати, що всі введені значення сприймаються як значення типу str, тобто рядки. І навіть якщо ми вводимо число, як у другому випадку в коді вище, то Python все одно розглядатиме введене значення як рядок, а не як число.

Керуючі послідовності в рядку.

Рядок може містити ряд спеціальних символів – послідовностей, що управляють ним. Деякі з них:

\\ - дає змогу додати всередину рядки слєш;

\' – дає змогу додати всередину рядка одинарну лапку;

\" – дає змогу додати всередину рядка подвійну лапку;

\n - здійснює перехід на новий рядок;

\t - додає табуляцію (4 відступи).

Застосуємо кілька послідовностей:

```
text = "Message:\n\"Hello World\""
print(text)
```

Хоча подібні послідовності можуть допомогти в деяких справах, наприклад, помістити в рядок лапку, зробити табуляцію, перенесення на інший рядок, проте вони також можуть заважати. Наприклад:

```
path = "C:\python\name.txt"
print(path)
```

Тут змінна *path* містить певний шлях до файлу. Однак усередині рядка зустрічається символ "\n", який буде інтерпретований як послідовність, що управляє.

Щоб уникнути подібної ситуації, перед рядком ставиться символ *r*:

```
path = r"C:\python\name.txt"
print(path)
```

Наприклад, ми хочемо, щоб усі значення виводилися на одному рядку. Для цього нам потрібно налаштувати поведінку функції за допомогою параметра *end*. Цей параметр визначає символи, які додаються в кінці до рядка. Під час застосування параметра *end* виклик функції *print()* виглядає так:

print(значення, що виводиться, end = кінцеві символи)

За замовчуванням *end* дорівнює символу *"\n"*, який задає перехід на наступний рядок. Власне тому функція *print()* за замовчуванням виводить значення, що їй передається, на окремому рядку.

Приклад коду, коли функція не робить переведення на наступний рядок, а виводить значення на тому самому рядку:

```
print("Hello World", end=" ")
print("Hello DPU.UA", end=" ")
print("Hello Python")
```

Тобто тепер значення, що виводяться, будуть розділятися пробілом:

```
Hello World Hello DPU.UA Hello Python
```

Причому це може бути не один символ, а набір символів:

```
print("Hello World", end=" and ")
print("Hello DPU.UA", end=" and ")
print("Hello Python")
```

У цьому випадку повідомлення, що виводяться, будуть розділятись символами *"and"*:

```
Hello World and Hello DPU.UA and Hello Python
```

Перелік питань для самоконтролю

1. Який є список ключових слів та інструкція для їх виведення?
2. Які є базові типи даних мови програмування Python?
3. Які є способи присвоювання значення змінним?
4. Функція перевірки типу даних.
5. Способи перетворення типів даних.
6. Способи видалення змінної.
7. Анотація змінних.
8. Консольне введення та виведення даних.
9. Коментарі.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Тема 4
Оператори в мові програмування Python

4.1. Різновидності математичних операторів та їх застосування.
4.2. Оператори належності.
4.3. Оператори присвоювання.
4.4. Пріоритет виконання операторів.
4.5. Оператори умови.
4.6. Оператори порівняння.
4.7. Логічні операці.

4.1. Різновидності математичних операторів та їх застосування

У Python існує кілька видів математичних операторів, які дають можливість виконувати різні математичні операції над числами та іншими об'єктами даних. Найпоширеніші математичні оператори в Python та їх застосування:

1. Арифметичні оператори:

- `+` (додавання) – додає два числа разом або об'єднує два рядки;
- `-` (віднімання) – віднімає одне число від іншого;
- `*` (множення – перемножує два числа або повторює рядок задану кількість разів;
- `/` (ділення) – ділить перше число на друге (з плавальною точкою);
- `//` (цілочисельне ділення) – ділить перше число на друге та повертає цілу частину результату;
- `%` (остача від ділення) – повертає остачу від ділення першого числа на друге;
- **`` (піднесення до степеня) – підносить перше число до степеня другого.

Під час послідовного використання кількох арифметичних операцій їх виконання здійснюється відповідно до пріоритету. Спочатку виконуються операції з великим пріоритетом.

Двійкові оператори. У Python існують двійкові оператори, які використовуються для виконання операцій над двійковими числами, тобто числами, представленими в двійковій системі числення. Приклади найпоширеніших двійкових операторів у Python та їх застосування.

1. Побітові оператори:

- `&` (Побітове І) – виконує побітову кон'юнкцію двох чисел;
- `|` (Побітове АБО) – виконує побітову диз'юнкцію двох чисел;
- `^` (Побітове виключне АБО) – виконує побітове виключне АБО (XOR) двох чисел;
- `~` (Побітова негація) – побітово інвертує всі біти (одноразове доповнення до двійкового представлення);
- `<<` (Зсув вліво) – зсуває біти числа вліво на задану кількість позицій (еквівалент множення на 2 у степені n);
- `>>` (Зсув вправо) – зсуває біти числа вправо на задану кількість позицій (еквівалент ділення на 2 у степені n).

2. Логічні двійкові оператори:

- `&` (Логічне І) – виконує логічне І над бітами двох чисел;
- `|` (Логічне АБО) – виконує логічне АБО над бітами двох чисел;
- `^` (Логічне виключне АБО) – виконує логічне виключне АБО (XOR) над бітами двох чисел.

3. Двійкові оператори присвоєння:

- `&=`, `|=`, `^=` (оператори присвоєння з побітовою операцією) – виконують відповідну побітову операцію над числами та присвоюють результат змінній.

Ці оператори дають змогу виконувати операції на рівні бітів у числах та здійснювати побітові маніпуляції з даними. Вони корисні для оптимізації деяких алгоритмів, роботи з прапорцями та здійснення інших операцій на низькому рівні.

4.2. Оператори належності

У Python оператори належності використовуються для перевірки належності елемента до певної множини або послідовності.

1. *in* – цей оператор перевіряє, чи входить елемент у послідовність (наприклад, рядок, список, кортеж або словник). Повертає *True*, якщо елемент перебуває в послідовності, і *False* в іншому випадку.

```
my_list = [1, 2, 3, 4, 5]
print(3 in my_list)  # Виведе True
print(6 in my_list)  # Виведе False
```

2. *not in* – цей оператор перевіряє, чи не входить елемент у послідовність. Повертає *True*, якщо елемент не перебуває в послідовності, і *False* в іншому випадку.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
print('d' not in my_dict)  # Виведе True
print('a' not in my_dict)  # Виведе False
```

3. *is* – цей оператор перевіряє, чи два об’єкти посилаються на один і той самий об’єкт у пам’яті. Повертає *True*, якщо об’єкти ідентичні, і *False* в іншому випадку.

```
x = [1, 2, 3]
y = x
print(x is y)  # Виведе True

z = [1, 2, 3]
print(x is z)  # Виведе False, тому що це різні об'єкти списків
```

Ці корисні для умовних виразів та керування потоком програми.

4.3. Оператори присвоювання

Оператори присвоювання в Python використовуються для присвоєння значень змінним.

1. = – просто присвоює праву частину виразу змінній лівої частини.

```
x = 10
```

2. += – додає праву частину виразу до значення змінної та присвоює результат змінній.

```
x += 5  # Еквівалентно x = x + 5
```

3. -= – віднімає праву частину виразу зі значення змінної та присвоює результат змінній.

```
y -= 3  # Еквівалентно y = y - 3
```

4. *= – перемножує значення змінної на праву частину виразу та присвоює результат змінній.

```
z *= 2  # Еквівалентно z = z * 2
```

5. /= – ділить значення змінної на праву частину виразу та присвоює результат змінній.

```
w /= 4  # Еквівалентно w = w / 4
```

6. %= – обчислює залишок від ділення значення змінної на праву частину виразу та присвоює результат змінній.

```
remainder %= 7  # Еквівалентно remainder = remainder % 7
```

7. //= – виконує цілочисельне ділення значення змінної на праву частину виразу та присвоює результат змінній.

```
quotient //= 2  # Еквівалентно quotient = quotient // 2
```

Ці оператори присвоювання дають змогу здійснювати операції над змінними та присвоювати результат обчислень тим самим змінним. Це зручно та спрощує написання коду.

4.4. Пріоритет виконання операторів

У Python оператори виконуються з різним пріоритетом, що впливає на порядок виконання виразів. Якщо ви використовуєте декілька операторів у виразі, вони виконуються відповідно до їх пріоритету.

1. Правило *PEMDAS* є абревіатурою, яка означає:

- дужки (Parentheses) – вирази всередині дужок обчислюються першими;
- піднесення до степеня (Exponent) – піднесення до степеня;
- множення (Multiplication) та ділення (Devision) – ці оператори мають однаковий пріоритет і виконуються зліва направо;
- додавання (Addition) та віднімання (Subtraction) – ці оператори також мають однаковий пріоритет і виконуються зліва направо.

2. Пріоритет вище – вищий пріоритет – оператори з більшим пріоритетом виконуються першими. Наприклад, множення має вищий пріоритет, ніж додавання, тому вираз 2 * 3 + 4 буде обчислено як (2 * 3) + 4, а не 2 * (3 + 4).

3. Групування операторів – ви можете використовувати дужки, щоб явно вказати порядок виконання виразів. Вирази всередині дужок обчислюються першими.

Нижче наведений приклад демонструє виконання операторів відповідно до їх пріоритету:

```
result = 5 * 3  2 + 10 / 2
# Спочатку виконується піднесення до степеня: 3  2 = 9
# Потім множення: 5 * 9 = 45
# Далі ділення: 10 / 2 = 5
# І нарешті додавання: 45 + 5 = 50
print(result)  # Виведе 50
```

4.5. Оператори умови

У Python оператори умови використовуються для визначення умови і виконання певних дій залежно від цієї умови.

1. *if* – використовується для виконання блоку коду, якщо задана умова є істинною.

```
x = 10
if x > 5:
    print("x більше за 5")
```

2. *if-else* – виконує блок коду, якщо умова істинна, і виконує інший блок коду, якщо умова є хибною.

```
y = 3
if y % 2 == 0:
    print("y є парним числом")
else:
    print("y є непарним числом")
```

3. *if-elif-else* – дає змогу перевірити кілька умов поспіль. Код виконується блоками, відповідно до першої істинної умови.

```
z = 0
if z > 0:
    print("z є додатним числом")
elif z < 0:
    print("z є від'ємним числом")
else:
    print("z дорівнює нулю")
```

4. Вкладені умовні оператори. Умовні оператори можуть бути вкладеними один в одного, щоб розгалуження було більш складним.

```
a = 30
if a > 10:
    if a % 2 == 0:
        print("a є парним числом, більшим за 10")
    else:
        print("a є непарним числом, більшим за 10")
```

Ці оператори умови дають змогу створювати гнучкі програми, які реагують на змінні умови та виконують певні дії відповідно до цих умов. Вони є невід'ємною частиною будь-якої мови програмування.

4.6. Оператори порівняння

У Python оператори порівняння використовуються для порівняння двох значень і повернення результату у формі логічного значення (*True* або *False*), залежно від того, чи задовольняє умова порівняння.

1. Рівність == – перевіряє, чи два значення рівні між собою.

```
x = 5
y = 5
result = (x == y)
print(result)  # Виведе True
```

2. Нерівність != – перевіряє, чи два значення не рівні між собою.

```
a = 10
b = 15
result = (a != b)
print(result)  # Виведе True
```

3. Більше > – перевіряє, чи перше значення більше за друге.

```
m = 20
n = 15
result = (m > n)
print(result)  # Виведе True
```

4. Менше < – перевіряє, чи перше значення менше за друге.

```
p = 10
q = 15
result = (p < q)
print(result)  # Виведе True
```

5. Більше або рівне >= – перевіряє, чи перше значення більше або рівне другому.

```
alpha = 10
beta = 10
result = (alpha >= beta)
print(result)  # Виведе True
```

6. Менше або рівне <= – перевіряє, чи перше значення менше або рівне другому.

```
gamma = 5
delta = 10
result = (gamma <= delta)
print(result)  # Виведе True
```

Ці оператори порівняння дають змогу порівнювати значення між собою та отримувати логічний результат, який вказує на відповідність умови порівняння. Вони широко використовуються для створення умовних виразів і керування поведінкою програми.

4.7. Логічні операції

У Python логічні операції використовуються для об'єднання, розділення та перевірки логічних умов. Ось кілька основних логічних операторів:

1. Логічне І (*and*) – логічне І повертає *True*, якщо обидва операнди є істинними. Якщо хоча б один з операндів є хибним, результат буде *False*.

```
x = 5
y = 10
result = (x < 10) and (y > 5)
print(result)  # Виведе True
```

2. Логічне АБО (*or*) – логічне АБО, повертає *True*, якщо хоча б один з операндів є істинним. Якщо обидва операнди є хибними, результат буде *False*.

```
a = 7
b = 12
result = (a < 5) or (b > 10)
print(result)  # Виведе True
```

3. Логічне НЕ (*not*) – логічне НЕ інвертує значення операнду. Якщо операнд є *True*, результат буде *False*, і навпаки.

```
flag = True
result = not flag
print(result)  # Виведе False
```

Ці логічні оператори дають змогу здійснювати перевірку складніших умов, об'єднуючи та перевіряючи логічні умови. Вони широко використовуються для побудови умовних виразів та контролю поведінки програми залежно від умов.

Перелік питань для самоконтролю

1. Математичні оператори в Python.
2. Двійкові оператори в Python.
3. Оператори належності в Python.
4. Пріоритет виконання операторів у Python.
5. Оператори умови в Python.
6. Оператори порівняння в Python.
7. Логічні операції в Python.
8. Логічні операції в Python.
9. Оператор in у Python.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть програму, яка приймає два числа від користувача й обчислює їхню суму, різницю, добуток і частку. Виведіть результати на екран.

2. Створіть програму, яка порівнює два числа, введені користувачем, і виводить на екран, чи є перше число більше, менше або дорівнює другому.

3. Напишіть програму, яка перевіряє, чи є введене користувачем число в межах від 10 до 20 (включно) і виводить відповідний результат за допомогою логічних операторів `and` та `or`.

4. Створіть програму, яка виконує такі операції над змінною `x`: присвоює їй значення 10, потім збільшує його на 5, потім зменшує на 3, і нарешті множить на 2. Виведіть остаточне значення `x`.

5. Напишіть програму, яка використовує оператори `+=` та `-=` для модифікації змінної `y`, спочатку додаючи 7, а потім віднімаючи 4. Виведіть результати після кожної операції.

6. Створіть програму, яка перевіряє, чи введене користувачем число є елементом списку `[3, 5, 7, 9, 11]`. Виведіть результат перевірки на екран.

7. Напишіть програму, яка перевіряє, чи дві змінні `a` і `b` посилаються на один і той самий об’єкт у пам’яті, використовуючи оператори `is` і `is not`. Виведіть відповідний результат.

8. Створіть програму, яка використовує оператори зсуву `<<` (зсув вліво) та `>>` (зсув вправо) для виконання зсуву бітів числа, введеного користувачем, на 2 позиції вліво і вправо відповідно. Виведіть результати.

9. Напишіть програму, яка використовує оператори `&` (бітове AND), `|` (бітове OR) та `^` (бітове XOR) для обробки двох цілих чисел, введених користувачем. Виведіть результати операцій на екран.

10. Створіть програму, яка приймає три числа від користувача і перевіряє, чи всі три числа різні, використовуючи оператори порівняння та логічні оператори. Виведіть результат перевірки на екран.

Тема 5
Оператори розгалуження в мові Python

5.1. Умовна конструкція if…else.
5.2. Вкладені конструкції if.
5.3. Умовна конструкція if…elif…else.

5.1. Умовна конструкція if…else

Умовна конструкція *if...else* використовується в Python для виконання певних блоків коду залежно від умови. Ця конструкція дає змогу програмі вирішувати, який блок коду виконувати, залежно від того, чи виконується задана умова.

Наведемо кілька прикладів використання умовної конструкції `if...else`:

1. Перевірка на парність числа:

```
num = 7

if num % 2 == 0:
    print(f"{num} є парним числом.")
else:
    print(f"{num} є непарним числом.")
```

2. Оцінка рівності двох чисел:

```
x = 5
y = 10

if x == y:
    print("Числа рівні.")
else:
    print("Числа не рівні.")
```

3. Порівняння двох чисел:

```
a = 15
b = 20

if a > b:
   print("a більше, ніж b.")
else:
   print("a менше або рівне b.")
```

4. Вибір між двома різними операціями:

```
temperature = 25

if temperature > 30:
   print("Занадто спекотно.")
else:
   print("Температура нормальна.")
```

5. Перевірка умови з використанням булевого значення:

```
is_raining = False

if is_raining:
   print("Взяти з собою парасольку.")
else:
   print("Погода гарна, можна піти гулять.")
```

У цих прикладах умовна конструкція *if...else* використовується для прийняття рішення щодо виконання певних дій на основі умови. Це дає змогу програмі реагувати на різні ситуації та виконувати відповідні дії залежно від цих ситуацій.

5.2. Вкладені конструкції if

Вкладені конструкції *if* використовуються в Python для визначення більш складних умов, де одна умова може бути залежною від іншої. Такі конструкції можуть мати багато рівнів вкладення, де кожен рівень перевіряє певну умову залежно від попереднього.

1. Перевірка декількох умов:

```
num = 75

if num > 0:
    if num % 2 == 0:
        print("Число є додатним парним числом.")
    else:
        print("Число є додатним непарним числом.")
else:
    print("Число менше або дорівнює нулю.")
```

2. Вибір між кількома опціями:

```
x = 5
y = 10

if x > y:
    print("x більше, ніж y.")
elif x < y:
    print("x менше, ніж y.")
else:
    print("x рівне y.")
```

3. Перевірка умов для різних типів даних:

```
value = input("Введіть число або рядок: ")

if value.isdigit():
   print("Введено число.")
elif value.isalpha():
   print("Введено рядок.")
else:
   print("Введено щось інше.")
```

4. Використання булевих значень для вкладення:

```
is_sunny = True
temperature = 25

if is_sunny:
   if temperature > 30:
      print("Варто піти на пляж.")
   else:
      print("Можна вийти на прогулянку.")
else:
   print("Краще залишитися вдома.")
```

У вищенаведених прикладах вкладені конструкції *if* використовуються для прийняття рішення в разі більш складних умов та виконання відповідних дій залежно від цих умов. Вкладеність дає змогу більш гнучко визначати, яку частину коду потрібно виконувати залежно від різних сценаріїв.

5.3. Умовна конструкція if…elif…else

Умовна конструкція *if...elif...else* використовується в Python для обробки багатьох умов, де потрібно перевіряти кілька варіантів умов та виконувати відповідні дії залежно від першої знайденої істинної умови.

1. Інтерпретація оцінок студентів:

```
mark = 75

if mark >= 90:
   print("Відмінно")
elif mark >= 80:
   print("Добре")
elif mark >= 70:
   print("Задовільно")
elif mark >= 60:
   print("Незадовільно")
else:
   print("Не склав")
```

2. Перевірка діапазону чисел:

```
num = 25

if num < 0:
   print("Від'ємне число")
elif 0 <= num <= 50:
   print("Число в діапазоні від 0 до 50")
elif 50 < num <= 100:
   print("Число в діапазоні від 51 до 100")
else:
   print("Число більше 100")
```

3. Вибір дії залежно від типу даних:

```
value = input("Введіть значення: ")

if value.isdigit():
   print("Ви ввели ціле число")
elif value.replace(".", "", 1).isdigit():
   print("Ви ввели дійсне число")
else:
   print("Ви ввели рядок")
```

4. Вибір між різними валютами:

```
currency = " EUR "

if currency == "USD":
   print("Долар США")
elif currency == "EUR":
   print("Євро")
elif currency == "GBP":
   print("Фунт стерлінгів")
else:
   print("Не відома валюта")
```

Перелік питань для самоконтролю

1. Умовна конструкція ***if…else.***
2. Вкладені конструкції ***if.***
3. Застосування умовної конструкції ***if…elif…else.***

Тестові питання

1. Яка конструкція використовується в Python для виконання блоку коду, якщо певна умова є істинною:

а) `for`;

б) `while`;

в) `if`;

г) `else`.

2. Що робить ключове слово `elif` у конструкції `if...elif...else`:

а) показує виконання блоку коду, якщо умова не виконується;

б) додає нову умову до конструкції `if`;

в) показує виконання блоку коду, якщо попередня умова не виконується, але поточна – так;

г) повертає значення True або False.

3. Що станеться, якщо в конструкції `if...else` жодна умова не виконується:

а) видасться помилка;

б) буде виконано блок коду після конструкції `else`;

в) виконання програми припиниться;

г) нічого не станеться, програма продовжить свою роботу.

4. Яка функція в Python використовується для перевірки типу даних:

а) `type()`;

б) `check_type()`;

в) `verify_type()`;

г) `data_type()`.

5. Що виконується, якщо в умові `if` вказано True:

а) блок коду виконується;

б) блок коду не виконується;

в) видається помилка;

г) потрібно використовувати `if...else`.

6. Який оператор використовується для перевірки рівності значень у Python:

а) `==`;

б) `=`;

в) `===`;

г) `!=`.

7. Яким буде результат виразу `10 > 5 and 20 < 15`:

а) True;

б) False;

в) None;

г) TypeError.

8. Що робить ключове слово `not` в умовному виразі:

а) перетворює значення на протилежне (True на False і навпаки);

б) перевіряє, чи два значення рівні;

в) виводить на екран результат;

г) виконує дію, якщо жодна умова не виконується.

9. Яка функція в Python використовується для перевірки істинності значень:

а) `is_true()`;

б) `check_truth()`;

в) `bool()`;

г) `verify_truth()`.

10. Яка конструкція використовується для обробки багатьох умов у Python:

а) `if...else`;

б) `while`;

в) `for`;

г) `if...elif...else`.

Рекомендовані джерела:

Основні: 1; 2; 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть програму, яка приймає вік користувача і перевіряє, чи він є студентом. Якщо вік менший за 18 років, то виведіть повідомлення "Студент молодшої школи". Якщо вік від 18 до 22 років, виведіть "Студент університету". Якщо більше 22 років, виведіть "Не студент".

2. Створіть програму, яка приймає оцінку і коефіцієнт (наприклад, 1.0 або 1.5) і обчислює кінцеву оцінку. Якщо кінцева оцінка більше або дорівнює 90, виведіть "Відмінно", якщо від 70 до 89 – "Добре", якщо менше 70 – "Задовільно".

3. Напишіть програму, яка перевіряє кількість проданих товарів. Якщо кількість більше 100, перевірте, чи продажі перевищують 200. Якщо так, виведіть "Високий рейтинг", інакше "Середній рейтинг". Якщо продажі менше або дорівнюють 100, виведіть "Низький рейтинг".

4. Створіть програму, яка приймає зріст користувача (у см) і вагу. Визначте розмір одягу: якщо зріст менший за 160 см, перевірте, чи вага менше 50 кг. Якщо так, виведіть "Малий розмір". Якщо вага більша або дорівнює 50 кг, виведіть "Середній розмір". Якщо зріст більший за 160 см, виведіть "Великий розмір".

5. Напишіть програму, яка приймає три оцінки від користувача. Якщо середня оцінка більша за 85, перевірте, чи всі оцінки більше 90. Якщо так, виведіть "Відмінник". Якщо середня оцінка від 70 до 85, виведіть "Хорошист". Якщо середня оцінка менше 70, виведіть "Неуспішний".

6. Створіть програму, яка перевіряє суму покупки. Якщо сума більша або дорівнює 1000 гривень, перевірте, чи є покупка більше 2000 гривень. Якщо так, виведіть "Знижка 20 %". Якщо сума менша 2000 гривень, виведіть "Знижка 10 %". Якщо сума менша 1000 гривень, виведіть "Знижки немає".

7. Напишіть програму, яка приймає температуру в градусах Цельсія. Якщо температура нижча за 0 градусів, перевірте, чи є температура нижча за -10 градусів. Якщо так, виведіть "Мороз". Якщо температура від 0 до 20 градусів, виведіть "Помірна температура". Якщо температура більше 20 градусів, виведіть "Тепло".

8. Створіть програму, яка перевіряє кількість товарів на складі. Якщо кількість менше 10 одиниць, перевірте, чи замовлення більше 50 одиниць. Якщо так, виведіть "Потрібно термінове замовлення". Якщо замовлення менше або дорівнює 50 одиниць, виведіть "Вистачає на деякий час". Якщо кількість товарів більше або дорівнює 10 одиниць, виведіть "Запасів достатньо".

9. Напишіть програму, яка приймає середній бал студента. Якщо середній бал більший або дорівнює 4.5, перевірте, чи є він стипендіатом. Якщо так, виведіть "Отримує стипендію". Якщо середній бал від 3.0 до 4.5, виведіть "Середній бал". Якщо менше 3.0, виведіть "Низький бал".

10. Створіть програму, яка перевіряє наявність дозволу на подорож. Якщо дозвіл є, перевірте, чи є додаткові вимоги (наприклад, візовий режим). Якщо є додаткові вимоги, виведіть "Необхідно подати документи". Якщо додаткових вимог немає, виведіть "Дозвіл на подорож є". Якщо дозволу немає, виведіть "Дозвіл на подорож не отримано".

11. Напишіть програму, яка приймає число від користувача і перевіряє, чи є воно парним чи непарним. Виведіть відповідне повідомлення на екран.

12. Створіть програму, яка приймає три числа від користувача і визначає найбільше з них за допомогою операторів `if`, `elif` та `else`. Виведіть результат на екран.

13. Напишіть програму, яка приймає оцінку від 0 до 100 і виводить рейтинг (наприклад, "A", "B", "C", "D", "F") відповідно до оцінки, використовуючи оператори розгалуження.

14. Створіть програму, яка перевіряє, чи введене користувачем число перебуває в діапазоні від 1 до 100 (включно). Виведіть відповідне повідомлення на екран.

15. Напишіть програму, яка приймає номер дня тижня (1-7) і виводить назву дня (понеділок, вівторок, тощо) за допомогою оператора `if-elif-else`.

16. Створіть програму, яка приймає вік користувача і визначає, чи є він дитиною (0–12 років), підлітком (13–19 років), дорослим (20–64 роки) або пенсіонером (65+ років). Виведіть відповідний результат.

17. Напишіть програму, яка перевіряє, чи є введене тризначне число паліндромом (тобто читається однаково з обох боків). Виведіть результат перевірки.

18. Створіть програму, яка перевіряє, чи містить введений користувачем рядок заданий символ. Виведіть відповідне повідомлення.

19. Напишіть програму, яка перевіряє, чи є введений рік високосним. Використовуйте правила високосного року (додавити високу кратність 4, але не кратність 100, за винятком кратності 400).

20. Створіть програму, яка приймає температуру в градусах Цельсія і виводить повідомлення, що описує погоду (наприклад, "холодно", "помірно", "жарко") на основі введеного значення температури.

Тема 6
Цикли

6.1. Оператор циклу for.
6.2. Оператор циклу while.
6.3. Вихід із циклу: break і continue.
6.4. Функції enumerate(), range(), len(), zip(), map().
6.5. Вкладені цикли в Python.

6.1. Оператор циклу for

Оператор циклу *for* використовується в Python для ітерації[3] за послідовністю об'єктів, таких як списки, кортежі, рядки тощо, та виконання певних дій з кожним елементом цієї послідовності. Синтаксис оператора циклу *for* такий:

for змінна in послідовність:
    блок коду, який виконується на кожній ітерації

Потрібно звернути увагу на відступ у другому рядку під оператором *for*.

У цьому синтаксисі *змінна* – це змінна, яка приймає значення кожного елемента з *послідовності*, а *послідовність* – це об'єкт, за яким буде виконуватися ітерація.

Приклади використання оператора циклу *for*:

1. Ітерація за списком чисел:

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)
```

[3] Ітера́ція (від лат. *Iteratio* – повторювання) — багатозначний термін, який залежно від контексту може означати повторне застосування математичної операції (зі зміненими даними) або будь-якої іншої під час розв'язання обчислювальних задач (або іншої), яке дає можливість поступово наблизитися до правильного результату.

2. Ітерація за рядком:

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

3. Ітерація за діапазоном чисел:

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

4. Ітерація за словником:

```
person = {'name': 'John', 'age': 30, 'city': 'New York'}
for key, value in person.items():
    print(f"{key}: {value}")
```

5. Використання оператора *break* для припинення циклу:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Оператор *break* дає змогу достроково перервати цикл:

- break перериває поточний цикл і продовжує виконання наступних виразів;
- якщо використовується кілька вкладених циклів, *break* перериває внутрішній цикл і продовжує виконувати вирази, що йдуть за блоком;
- break може використовуватися в циклах *for* та *while.*

На відміну від *continue*, який пропускає лише одну ітерацію циклу, оператор *break* повністю виходить з циклу.

6. Використання оператора *continue* для переходу до наступної ітерації:

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Оператор *continue* повертає керування на початок циклу. Тобто *continue* дає змогу "перестрибнути" вирази, що залишилися в циклі, і перейти до наступної ітерації.

6.2. Оператор циклу while

Оператор циклу *while* використовується в Python для повторення блоку коду доти, доки певна умова залишається істинною. Коли умова стає хибною, виконання циклу припиняється і виконується наступний блок коду після циклу.

1. Виведення чисел від 1 до 5:

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

2. Підрахунок суми перших 10 натуральних чисел:

```
total = 0
i = 1
while i <= 10:
    total += i
    i += 1
print("Сума перших 10 натуральних чисел:", total)
```

3. Введення користувачем числа, доки воно не буде більше ніж 100:

```
num = int(input("Введіть число: "))
while num <= 100:
    num = int(input("Введіть число (ще раз): "))
print("Ви ввели число більше за 100.")
```

4. Гра "Вгадай число":

```
import random

number_to_guess = random.randint(1, 100)
guess = None

while guess != number_to_guess:
    guess = int(input("Спробуйте вгадати число (від 1 до 100): "))
    if guess < number_to_guess:
        print("Загадане число більше.")
    elif guess > number_to_guess:
        print("Загадане число менше.")
    else:
        print("Ви вгадали!")
```

5. Використання оператора *break* для припинення циклу:

```
i = 1
while i <= 10:
    print(i)
    if i == 5:
        break
    i += 1
```

6. Використання оператора *continue* для переходу до наступної ітерації:

```
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    print(i)
```

6.3. Вихід із циклу: break і continue

Оператори *break* і *continue* використовуються в мові програмування Python для керування виконанням циклів.

- *break* – використовується для негайного припинення виконання циклу і виходу з нього;

- *continue* – використовується для переходу безпосередньо до наступної ітерації циклу, припускаючи, що далі код після *continue* буде проігноровано.

1. Вихід з циклу в разі досягнення певної умови з оператором *break*:

```
i = 1
while i <= 10:
    print(i)
    if i == 5:
        break
    i += 1
```

2. Пропуск ітерації циклу за певної умови з оператором *continue*:

```
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    print(i)
```

3. Використання *break* для виходу з циклу за певної умови вкладеного циклу:

```
for i in range(3):
    for j in range(3):
        print(i, j)
        if j == 1:
            break  # Вихід з внутрішнього циклу
```

4. Використання *continue* для пропуску певної умови вкладеного циклу:

```
for i in range(3):
    for j in range(3):
        if j == 1:
            continue  # Пропустити поточну ітерацію
        print(i, j)
```

6.4. Функції enumerate(), range(), len(), zip(), map()

1. *enumerate(iterable, start=0)* – використовується для нумерації елементів ітерабельного об’єкта, повертає ітератор, що містить кортежі `(індекс, значення)`.

```
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits, start=1):
    print(f"#{index}: {fruit}")
```

2. *range(start, stop, step)* – генерує послідовність чисел у заданому діапазоні, повертає об’єкт діапазону, який можна перетворити у список або ітерувати.

```
for i in range(5):
    print(i)
# Використання кроку
for i in range(1, 10, 2):
    print(i)
```

3. *len(sequence)* – повертає кількість елементів у послідовності.

```
my_list = [10, 20, 30, 40, 50]
print("Кількість елементів у списку:", len(my_list))
```

4. zip(*iterables) – об'єднує елементи з кількох ітерабельних об'єктів у кортежі, повертає ітератор, який генерує кортежі з відповідними елементами з кожного вхідного ітерабельного об'єкта.

```
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
```

5. *map(function, iterable)* – використовується для виконання функції на кожному елементі ітерабельного об'єкта, повертає ітератор, що містить результати застосування функції до кожного елемента.

```
def square(x):
    return x  2

numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)
print(list(squared_numbers))
```

Ці функції дають змогу ефективно керувати ітерацією по об'єктах, вимірювати довжину послідовностей, об'єднувати елементи з різних ітерабельних об'єктів та виконувати функції на кожному елементі ітерабельного об'єкта.

6.5. Вкладені цикли в Python

Владені цикли – це коли один цикл міститься всередині іншого циклу. Це дає змогу Вам виконувати певний блок коду декілька разів для кожного елемента у внутрішньому циклі, використовуючи кожен елемент у зовнішньому циклі.

1. Приклад вкладеного циклу *for*:

```
for i in range(3): # Зовнішній цикл
    for j in range(2): # Внутрішній цикл
        print(i, j)
```

У цьому прикладі зовнішній цикл *for* повторюється 3 рази, а внутрішній цикл *for* повторюється 2 рази для кожного ітератора зовнішнього циклу. Результатом буде:

```
0 0
0 1
1 0
1 1
2 0
2 1
```

2. Приклад вкладеного циклу *while*:

```
i = 1
while i <= 3: # Зовнішній цикл
    j = 1
    while j <= 2: # Внутрішній цикл
        print(i, j)
        j += 1
    i += 1
```

У цьому прикладі зовнішній цикл *while* повторюється, доки *i* менше або дорівнює 3, і внутрішній цикл *while* повторюється, доки *j* менше або дорівнює 2, для кожного значення *i* у зовнішньому циклі. Результат буде такий самий, як і в попередньому прикладі.

3. Вкладений цикл *for* для роботи зі списками списків:

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

for row in matrix: # Зовнішній цикл
    for item in row: # Внутрішній цикл
        print(item)
```

У цьому прикладі ми маємо список списків *matrix*. Зовнішній цикл *for* ітерується по кожному списку у *matrix*, а внутрішній цикл *for* ітерується по кожному елементу в кожному списку. Результатом буде виведення кожного елемента списку:

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

Це лише кілька прикладів вкладених циклів у Python. Вони можуть бути корисними для різних завдань, таких як обхід елементів матриці або виконання певних дій для кожного елемента вкладеної структури даних.

Перелік питань для самоконтролю

1. Вкладені цикли в Python.
2. Застосування циклу ***for*** у Python.
3. Застосування циклу ***while*** в Python.
4. Застосування ***break*** і ***continue*** в Python.
5. Функції ***enumerate(), range(), len(), zip(), map()*** у Python.

Тестові питання

1. Що таке вкладений цикл:

а) цикл, який виконується виключно в Python;

б) цикл, який містить у собі інші цикли;

в) цикл, який виконується безперервно;

г) цикл, який виконується лише один раз.

2. Який оператор використовується для створення циклу `for` у Python:

а) `do`;
б) `foreach`;
в) `for`;
г) `while`.

3. Як виконати певні дії певну кількість разів у Python:

а) використати цикл `loop`;
б) використати цикл `for`;
в) використати команду `repeat`;
г) використати цикл `if`.

4. Що робить функція `range()` у циклі `for` у Python:

а) установлює діапазон значень для ітерації;
б) визначає кількість повторень циклу;
в) виконує цикл для списку значень;
г) визначає кінцеве значення циклу.

5. Яка конструкція використовується для виходу з циклу довільно в Python:

а) `halt`;
б) `return`;
в) `stop`;
г) `break`.

6. Яке ключове слово використовується для переходу до наступної ітерації циклу в Python:

а) `skip`;
б) `continue`;
в) `next`;
г) `pass`.

7. Які два типи циклів підтримуються в Python:

а) `for` і `loop`;
б) `while` і `foreach`;
в) `for` і `while`;
г) `do` і `while`.

8. Який оператор використовується для створення циклу `while` в Python:

а) `for`;
б) `do`;
в) `until`;
г) `while`.

9. Який оператор потрібно використовувати, щоб створити вкладений цикл у Python:

а) `for`;
б) `foreach`;
в) `while`;
г) `loop`.

10. Що буде результатом виконання коду `for i in range(5): print(i)` у Python:

а) 0 1 2 3 4;
б) 1 2 3 4 5;
в) 0 1 2 3;
г) 1 2 3 4.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть програму, яка використовує цикл `while` для виведення чисел від 1 до 10 на екран.

2. Створіть програму, яка приймає число `N` від користувача і використовує цикл `while` для обчислення суми всіх чисел від 1 до `N` (включно).

3. Напишіть програму, яка використовує цикл `while` для виведення всіх парних чисел у діапазоні від 1 до 20.

4. Створіть програму, яка використовує цикл `while` для перевірки, чи є введене число простим (ділиться тільки на 1 і на саме себе).

5. Напишіть програму, яка обчислює факторіал числа (n!) за допомогою циклу `while`. Введіть число від користувача і виведіть його факторіал.

6. Створіть програму, яка використовує цикл `while`, щоб продовжувати запитувати користувача ввести число, поки він не введе позитивне число. Виведіть введене позитивне число.

7. Напишіть програму, яка приймає рядок від користувача і використовує цикл `while` для виведення символів цього рядка в зворотному порядку.

8. Створіть програму, яка використовує цикл `while` для додавання чисел до списку доти, доки сума елементів списку не перевищить 100. Виведіть список після завершення циклу.

9. Напишіть програму, яка використовує цикл `while` для виведення чисел, що є ступенями двійки (2, 4, 8, 16 тощо), доки число менше або дорівнює 1000.

10. Створіть програму, яка використовує цикл `while` для запитування пароля в користувача доти, доки він не введе правильний пароль (наприклад, "password123"). Виведіть повідомлення про успішний вхід після правильного введення пароля.

11. Напишіть програму, яка використовує цикл `for` для обчислення суми всіх цілих чисел від 1 до 10 (включно) і виводить результат.

12. Створіть програму, яка виводить усі парні числа в діапазоні від 1 до 20 (включно) за допомогою циклу `for`.

13. Напишіть програму, яка обчислює факторіал числа (n!) за допомогою циклу `for`. Введіть число від користувача і виведіть його факторіал.

14. Створіть програму, яка використовує цикл `for` для створення списку квадратів чисел від 1 до 10 (включно) і виводить цей список.

15. Напишіть програму, яка приймає рядок від користувача і використовує цикл `for`, щоб вивести кожен символ цього рядка на окремому рядку.

16. Створіть програму, яка приймає список чисел і використовує цикл `for` для виведення елементів списку в зворотному порядку.

17. Напишіть програму, яка використовує цикл `for` для обчислення суми всіх чисел у заданому списку (наприклад, `[5, 10, 15, 20]`) і виводить результат.

18. Створіть програму, яка перевіряє, чи є число простим (тобто ділиться тільки на 1 і на саме себе) за допомогою циклу `for`. Виведіть повідомлення, чи є число простим.

19. Напишіть програму, яка використовує цикл `for` для створення таблиці множення для числа, введеного користувачем, від 1 до 10. Виведіть таблицю у вигляді таблички.

20. Створіть програму, яка приймає два числа від користувача: `n` та `m`. Використовуйте цикл `for` для створення списку чисел, які є кратними `n` і меншими за `m`. Виведіть цей список.

21. Напишіть програму, яка створює квадратну матрицю розміру `n x n`, де `n` задається користувачем. Заповніть матрицю числами від 1 до `n^2` у порядку зростання.

22. Створіть програму, яка генерує таблицю множення від 1 до 10. Виведіть результати у вигляді таблиці, де кожен рядок представляє результати множення одного числа від 1 до 10 на всі числа від 1 до 10.

23. Згенеруйте таблицю факторіалів для чисел від 1 до `n`, де `n` задається користувачем. Результати повинні бути представлені у вигляді таблиці, де кожен рядок показує число і його факторіал.

24. Напишіть програму, яка виводить числа від 1 до 100 у формі сітки 10 x 10, де кожне число відображається у форматі з шириною 4 пробіли.

25. Напишіть програму, яка знаходить та виводить елементи діагоналей головної та побічної діагоналі квадратної матриці розміру `n x n`.

26. Напишіть програму, яка запитує користувача ввести числа й обчислює їхню суму. Якщо користувач вводить число `0`, цикл повинен зупинитися, а програма повинна вивести суму всіх введених чисел (крім нуля).

27. Напишіть програму, яка виводить усі парні числа від 1 до 20. Використовуйте `continue`, щоб пропускати непарні числа.

28. Створіть програму, яка знаходить прості числа в діапазоні від 2 до `n` (де `n` задається користувачем). Якщо число не є простим, пропустіть його і продовжуйте перевірку наступного числа.

29. Напишіть програму, яка запитує користувача ввести числа й обчислює їхню середню величину. Якщо користувач вводить не число (наприклад, текст), програма повинна зупинити цикл і вивести повідомлення про помилку.

30. Напишіть програму, яка запитує користувача ввести кілька чисел, допоки не буде введене число, яке є парним. Продовжуйте запитувати числа, доки користувач не введе парне число, і виведіть це число.

Тема 7
Базові типи для представлення чисел

7.1. Представлення чисел у десятковій, двійковій, вісімковій та шістнадцятковій системах числення.

7.2. Числа, що задані з фіксованою точністю за підтримки модуля decimal.

7.3. Виконання операцій над дробами за підтримки методів модуля fractions.

7.4. Вбудовані функції для роботи з числами.

7.5. Стандартні константи модуля math.

7.6. Формат представлення та операції над комплексними числами.

7.7. Застосування модуля random для генерації випадкових чисел.

7.1. Представлення чисел у десятковій, двійковій, вісімковій та шістнадцятковій системах числення

1. Представлення числа в десятковій системі числення:

```
decimal_number = 42
print("Десяткове число:", decimal_number)
```

2. Представлення того самого числа у двійковій системі числення:

```
binary_number = bin(decimal_number)
print("Двійкове число:", binary_number)
```

3. Представлення того самого числа у вісімковій системі числення:

```
octal_number = oct(decimal_number)
print("Вісімкове число:", octal_number)
```

4. Представлення того самого числа у шістнадцятковій системі числення:

```
hexadecimal_number = hex(decimal_number)
print("Шістнадцяткове число:", hexadecimal_number)
```

Ці приклади виведуть таке:

```
Десяткове число: 42
Двійкове число: 0b101010
Вісімкове число: 0o52
Шістнадцяткове число: 0x2a
```

У Python функції *bin()*, *oct()* і *hex()* використовуються для представлення чисел у відповідно двійковій, вісімковій та шістнадцятковій системах числення.

7.2. Числа, що задані з фіксованою точністю за підтримки модуля decimal

Модуль *decimal* у Python дає змогу працювати з числами, що мають фіксовану точність, що є корисним для точних обчислень, де точність є важливою.

1. Імпорт модуля та створення числа з фіксованою точністю:

```
from decimal import Decimal

# Створення числа з фіксованою точністю
fixed_number = Decimal('10.5')
print(fixed_number)
```

2. Виконання арифметичних операцій з числами з фіксованою точністю:

```
# Додавання чисел з фіксованою точністю
result = fixed_number + Decimal('20.75')
print(result)

# Множення числа з фіксованою точністю
result = fixed_number * Decimal('3')
print(result)
```

3. Встановлення точності для арифметичних операцій:

```
# Встановлення точності до 3 знаків після коми
Decimal.getcontext().prec = 3

result = fixed_number / Decimal('3')
print(result)
```

Метод *getcontext().prec* установлює точність для арифметичних операцій.

7.3. Виконання операцій над дробами за підтримки методів модуля fractions

Модуль *fractions* у Python дає змогу працювати з дробами та виконувати над ними різні арифметичні операції.

1. Імпорт модуля та створення дробу:

```
from fractions import Fraction

# Створення дробу
frac1 = Fraction(1, 3)
frac2 = Fraction(2, 5)
print("Дріб 1:", frac1)
print("Дріб 2:", frac2)
```

2. Додавання дробів:

```
# Додавання дробів
result = frac1 + frac2
print("Сума дробів:", result)
```

3. Віднімання дробів:

```
# Віднімання дробів
result = frac1 - frac2
print("Різниця дробів:", result)
```

4. Множення дробів:

```
# Множення дробів
result = frac1 * frac2
print("Добуток дробів:", result)
```

5. Ділення дробів:

```
# Ділення дробів
result = frac1 / frac2
print("Частка дробів:", result)
```

Методи `+`, `-`, `*` та `/` використовуються для відповідних арифметичних операцій над дробами.

7.4. Вбудовані функції для роботи з числами

У Python є кілька вбудованих функцій для роботи з числами, які допомагають виконувати різноманітні операції, від округлення чисел до знаходження максимального або мінімального значення.

1. *abs()* – повертає абсолютне значення числа, тобто його відстань від нуля:

```
number = -10
absolute_value = abs(number)
print(absolute_value)  # Виведе: 10
```

2. *round()* – округлює число до заданої кількості десяткових знаків:

```
number = 3.14159
rounded_number = round(number, 2)
print(rounded_number)  # Виведе: 3.14
```

3. *max()* – повертає найбільше число з набору чисел:

```
numbers = [5, 8, 2, 10]
max_value = max(numbers)
print(max_value)  # Виведе: 10
```

4. *min()* – повертає найменше число з набору чисел:

```
numbers = [5, 8, 2, 10]
min_value = min(numbers)
print(min_value)  # Виведе: 2
```

5. *sum()* – обчислює суму всіх чисел у списку:

```
numbers = [1, 2, 3, 4, 5]
sum_of_numbers = sum(numbers)
print(sum_of_numbers)  # Виведе: 15
```

6. *pow()* – підносить число до заданого степеня:

```
base = 2
exponent = 3
result = pow(base, exponent)
print(result)  # Виведе: 8
```

7. *divmod()* – повертає частку та залишок від ділення двох чисел:

```
quotient, remainder = divmod(10, 3)
print("Частка:", quotient)  # Виведе: 3
print("Залишок:", remainder)  # Виведе: 1
```

Ці вбудовані функції є корисними для різноманітних операцій з числами, що допомагають спростити обробку даних та виконання обчислень.

7.5. Стандартні константи модуля math

Модуль *math* у Python надає доступ до ряду математичних функцій і констант. Однією зі стандартних констант у модулі є значення π, яке представлене як math.pi. Це лише одна з констант, доступних у модулі *math*.

```
import math

# Значення pi (π)
print("Значення pi (π):", math.pi)

# Значення e (основа натурального логарифму)
print("Значення e:", math.e)
```

Приклади виведуть такий результат:

```
Значення pi (π): 3.141592653589793
Значення e: 2.718281828459045
```

Модуль *math* у Python містить багато методів та функцій для виконання різноманітних математичних операцій.

1. *math.sqrt(x)* – обчислює квадратний корінь числа `x`:

```
import math

result = math.sqrt(16)
print(result)  # Виведе: 4.0
```

2. *math.pow(x, y)* – підносить число `*x*` до степеня `*y*`:

```
import math

result = math.pow(2, 3)
print(result)  # Виведе: 8.0
```

3. *math.exp(x)* – обчислює експоненту числа `*x*`:

```
import math

result = math.exp(1)
print(result)  # Виведе: 2.718281828459045
```

4. *math.log(x[, base])* – обчислює натуральний логарифм числа `*x*`. Опціонально, можна вказати основу логарифма:

```
import math

result = math.log(10)
print(result)  # Виведе: 2.302585092994046
```

5. *math.sin(x), math.cos(x), math.tan(x)* – обчислюють синус, косинус і тангенс кута `*x*` (у радіанах):

```
import math

sin_value = math.sin(math.pi / 2)
cos_value = math.cos(0)
tan_value = math.tan(math.pi / 4)

print("Синус:", sin_value)  # Виведе: 1.0
print("Косинус:", cos_value)  # Виведе: 1.0
print("Тангенс:", tan_value)  # Виведе: 0.9999999999999999
```

6. *math.radians(x), math.degrees(x)* – перетворюють кут `*x*` з градусів у радіани і навпаки:

```
import math

angle_in_degrees = 90
angle_in_radians = math.radians(angle_in_degrees)

print("Кут у радіанах:", angle_in_radians)  # Виведе:
1.5707963267948966

angle_in_degrees = math.degrees(math.pi / 2)
print("Кут у градусах:", angle_in_degrees)  # Виведе: 90.0
```

Ці функції та методи лише частина того, що пропонує модуль `math`. Він також містить функції для роботи з тригонометричними функціями, гіперболічними функціями, константами та багато іншого.

7.6. Формат представлення та операції над комплексними числами

У Python комплексні числа представляються у формі `*a* + *bj*`, де `*a*` та `*b*` – дійсна та уявна частини відповідно, а `*j*` – уявна одиниця. Операції над комплексними числами можуть бути виконані за допомогою вбудованих функцій та методів.

1. Створення комплексного числа:

```
# Створення комплексного числа (3 + 4j)
z = 3 + 4j
print(z)  # Виведе: (3+4j)
```

2. Доступ до дійсної та уявної частини комплексного числа:

```
z = 3 + 4j
real_part = z.real
imaginary_part = z.imag
print("Дійсна частина:", real_part)  # Виведе: 3.0
print("Уявна частина:", imaginary_part)  # Виведе: 4.0
```

3. Додавання, віднімання, множення та ділення комплексних чисел:

```
z1 = 2 + 3j
z2 = 1 - 2j

# Додавання
result_addition = z1 + z2
print("Додавання:", result_addition)  # Виведе: (3+1j)

# Віднімання
result_subtraction = z1 - z2
print("Віднімання:", result_subtraction)  # Виведе: (1+5j)

# Множення
result_multiplication = z1 * z2
print("Множення:", result_multiplication)  # Виведе: (8-1j)

# Ділення
result_division = z1 / z2
print("Ділення:", result_division)  # Виведе: (0.8+1.4j)
```

4. Модуль комплексного числа (абсолютне значення):

```
z = 3 + 4j
absolute_value = abs(z)
print("Модуль комплексного числа:", absolute_value)  # Виведе: 5.0
```

5. Кон’югат комплексного числа (заміна уявної частини на протилежне значення):

```
z = 3 + 4j
conjugate = z.conjugate()
print("Кон'югат комплексного числа:", conjugate)  # Виведе: (3-4j)
```

Це лише деякі з операцій, які можна виконати над комплексними числами в Python. Комплексні числа можуть бути використані для розв’язання різноманітних математичних проблем, таких як робота з функціями аналізу, фізики або інженерії.

7.7. Застосування модуля random для генерації випадкових чисел

Модуль *random* у Python надає можливість генерувати випадкові числа та виконувати різноманітні операції, пов’язані з випадковістю.

1. *random.random()* – генерує випадкове число в діапазоні від 0 до 1.

```
import random

random_number = random.random()
print(random_number)  # Виведе випадкове число, наприклад: 0.7526575342631065
```

2. *random.randint(a, b)* – генерує випадкове ціле число в діапазоні від *a* до *b* (включно).

```
import random

random_integer = random.randint(1, 100)
print(random_integer)  # Виведе випадкове ціле число від 1 до 100
```

3. *random.choice(seq)* – обирає випадковий елемент зі списку *seq*.

```
import random

my_list = ['apple', 'banana', 'cherry', 'date']
random_item = random.choice(my_list)
print(random_item)  # Виведе випадковий елемент зі списку
```

4. *random.shuffle(x)* – перемішує послідовність *x* у випадковому порядку.

```
import random

my_list = [1, 2, 3, 4, 5]
random.shuffle(my_list)
print(my_list)  # Виведе послідовність, перемішану у випадковому порядку
```

5. *random.sample(population, k)* – повертає список довжиною *k*, який містить випадкові елементи з *population* без повторень.

```
import random

my_list = [1, 2, 3, 4, 5]
random_sample = random.sample(my_list, 3)
print(random_sample)  # Виведе список з 3 випадковими елементами зі списку, наприклад: [2, 4, 1]
```

Перелік питань для самоконтролю

1. Операції над комплексними числами.
2. Модуль *decimal*.
3. Основні функції модуля *math*.
4. Модуль *random* для генерації випадкових чисел.

Тестові питання

1. Який з наступних базових типів використовується для представлення цілих чисел у Python:

а) `int`;
б) `float`;
в) `complex`;
г) `str`.

2. Які з нижченаведених чисел є прикладами чисел типу `float` у Python:

а) 10;
б) 3.14;
в) '42';
г) (1 + 2j).

3. Які з наступних операторів можуть бути використані для виконання арифметичних операцій над числами типу `float` у Python:

а) `+`;
б) `-`;
в) `*`;
г) усі відповіді правильні.

4. Який тип даних використовується для представлення комплексних чисел у Python:

а) `complex`;
б) `int`;
в) `float`;
г) `str`.

5. Які з наступних виразів є прикладами комплексних чисел у Python:

а) 42;
б) 3.14;
в) '2 + 3j';
г) (1 - 2j).

6. Який тип даних використовується для представлення раціональних чисел у Python:

а) `int`;

б) `float`;

в) `complex`;

г) `Fraction`.

7. Які з наступних методів можуть бути використані для перетворення рядка у число у Python:

а) `str()`;

б) `int()`;

в) `float()`;

г) усі відповіді правильні.

8. Які з наступних операцій будуть неможливими для виконання з числами типу `int` у Python:

а) додавання;

б) віднімання;

в) множення;

г) ділення на інше ціле число.

9. Який з наступних операторів може бути використаний для піднесення числа до певного степеня в Python:

а) ``;

б) `^`;

в) `^*`;

г) `pow()`.

10. Яка функція в Python може використовуватися для знаходження модуля числа:

а) `abs()`;

б) `mod()`;

в) `module()`;

г) `absolute()`.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть програму, яка запитує користувача ввести число в форматі рядка. Перетворіть цей рядок спочатку в `int`, потім у `float` і виведіть обидва значення разом з їх типами.

2. Напишіть програму, яка приймає три числа від користувача (у форматі `int` або `float`) й обчислює їх середнє значення. Виведіть результат.

3. Створіть програму, яка приймає два комплексних числа у форматі `a+bj` від користувача, додає їх і виводить результат.

4. Напишіть програму, яка запитує в користувача два числа: одне у форматі `int`, інше у форматі `float`. Продовжте обчислення їхньої суми як `float` та виведіть результат.

5. Створіть програму, яка перевіряє, чи є введене число (як `float`) цілим. Виведіть відповідне повідомлення.

6. Напишіть програму, яка приймає комплексне число й розділяє його на реальну та уявну частини, виводячи їх окремо.

7. Створіть програму, яка запитує в користувача число з плаваючою комою і виводить це число, округлене до найближчого цілого, до двох десяткових знаків і до трьох десяткових знаків.

8. Напишіть програму, яка запитує в користувача число й обчислює його квадрат і квадратний корінь. Виведіть результати.

9. Напишіть програму, яка перевіряє, чи є введене число (як `float`) складеним числом (число, більше 1, яке не є простим).

10. Напишіть програму, яка перевіряє, чи є введене число від'ємним, нульовим або додатним, і виводить відповідне повідомлення.

11. Напишіть програму, яка запитує у користувача число і перевіряє, чи є це число парним або непарним, використовуючи оператор `%`.

12. Напишіть програму, яка запитує в користувача число і виводить його абсолютне значення, використовуючи функцію `abs()`.

13. Напишіть програму, яка приймає три числа від користувача і використовує функції `max()` і `min()` для знаходження найбільшого та найменшого числа серед введених.

14. Створіть програму, яка запитує в користувача число з плаваючою комою і округляє його до найближчого цілого, використовуючи функцію `round()`.

15. Напишіть програму, яка приймає число від користувача і використовує функцію `math.sqrt()` для обчислення його квадратного кореня.

16. Створіть програму, яка запитує в користувача число, перетворює його в рядок, а потім повертає назад у число і виводить його.

17. Напишіть програму, яка перевіряє, чи є введене число простим, використовуючи цикл і функцію `range()`.

18. Створіть програму, яка приймає ціле число від користувача і використовує функцію `math.factorial()` для обчислення його факторіала.

19. Напишіть програму, яка перевіряє, чи є введене число степенем двійки, використовуючи вбудовані функції.

20. Напишіть програму, яка запитує в користувача число з плаваючою комою і виводить його з точністю до трьох десяткових знаків, використовуючи форматування рядків.

Тема 8
Рядки

8.1. Оголошення рядка.
8.2. Операції над рядками.
8.3. Екрановані послідовності.
8.4. Неформатовані рядки.
8.5. Багаторядкові блоки тексту.
8.6. Доступ за індексами.
8.7. Зрізи.
8.8. Отримання фрагменту рядка.
8.9. Засоби перетворення рядків та одиночних символів. Функції int(), str(), repr(), float(), ord(), chr().
8.10. Функції len(), max(), min().
8.11. Доступ за індексами.

8.1. Оголошення рядка

У Python рядки можна оголосити, використовуючи одинарні (' '), подвійні (" ") або потрійні ('" "' або """ """) лапки.

1. Одинарні лапки:

```
my_string = 'Це рядок у одинарних лапках'
print(my_string)
```

2. Подвійні лапки:

```
my_string = "Це рядок у подвійних лапках"
print(my_string)
```

3. Потрійні лапки:

```
my_string = '''Це рядок
у потрійних лапках'''
print(my_string)
```

4. Змішане використання лапок:

```
my_string = "Це рядок у 'подвійних лапках', але зі 'строковою' частиною"
print(my_string)
```

Використання потрійних лапок дає змогу оголошувати багатострокові рядки та включати в них апострофи й подвійні лапки без необхідності в екрануванні. Крім того, рядки в подвійних та одинарних лапках можуть включати символи нового рядка, але в потрійних лапках це може бути зроблено безпосередньо, без використання спеціальних символів.

8.2. Операції над рядками

У Python існує багато операцій, які можна виконати над рядками.

1. Конкатенація рядків (об’єднання):

```
string1 = "Привіт"
string2 = ", світ!"
result = string1 + string2
print(result)  # Виведе: Привіт, світ!
```

2. Повторення рядка (множення на ціле число):

```
string = "Python"
result = string * 3
print(result)  # Виведе: PythonPythonPython
```

3. Доступ до символів та підрядків:

```
string = "Python"
print(string[0])    # Виведе: P
print(string[1:4])  # Виведе: yth
```

4. Перевірка наявності підрядка в рядку:

```
string = "Python programming"
substring = "gram"
if substring in string:
   print("Підстрока знайдена")
else:
   print("Підстрока не знайдена")
```

5. Зміна регістру рядка:

```
string = "Python"
print(string.lower())  # Виведе: python
print(string.upper())  # Виведе: PYTHON
```

6. Розділення рядка на список підрядків:

```
string = "Python programming is fun"
words = string.split()
print(words)  # Виведе: ['Python', 'programming', 'is', 'fun']
```

7. Заміна підрядка в рядку:

```
string = "Hello, world!"
new_string = string.replace("world", "Python")
print(new_string)  # Виведе: Hello, Python!
```

8. Видалення пробілів з початку та кінця рядка:

```
string = "  Hello, world!  "
trimmed_string = string.strip()
print(trimmed_string)  # Виведе: Hello, world!
```

Операції над рядками в Python можна розділити на кілька категорій, таких як операції конкатенації, операції доступу до елементів рядка, операції порівняння, операції пошуку, операції заміни та форматування, операції з регістром та інші.

1. Конкатенація (+) – об’єднання двох рядків.

2. Повторення (*) – повторює рядок задану кількість разів.

3. Доступ до елементів та підрядків:

- [] – доступ до символу за індексом;

- [:] – отримання підрядка за зрізом.

4. Перевірка наявності підрядка (*in*) – перевірка наявності підрядка в рядку.

5. Заміна та форматування:

- *replace()* – заміна підрядка іншим підрядком;

- *format()* – форматування рядка з використанням підстановки значень.

6. *split()* – розділення рядка на список підрядків за розділь-ником.

7. Пошук:

- *find()* – пошук позиції першого входження підрядка.

- *index()* – пошук позиції першого входження підрядка, але видає помилку, якщо підрядок не знайдено.

8. Регістр:

- *lower()* – переведення рядка в нижній регістр;

- *upper()* – переведення рядка у верхній регістр;

- *capitalize()* – перша літера рядка стає великою, а всі інші – малими.

9. Видалення пробілів:

- *strip()* – видалення пробілів з початку та кінця рядка;

- *lstrip()* – видалення пробілів з початку рядка;

- *rstrip()* – видалення пробілів з кінця рядка.

10. *len()* – повертає довжину рядка.

11. `<`, `<=`, `>`, `>=`, `==`, `!=` – порівняння двох рядків.

12. *%* – форматування рядка за допомогою оператора.

13. *str()* – перетворення об’єкта в рядок.

14. Інші:

- *startswith()* – перевірка, чи рядок починається з певного підрядка.

- *endswith()* – перевірка, чи рядок закінчується певним під-рядком.

8.3. Екрановані послідовності

Екрановані послідовності в Python – це спеціальні символьні послідовності, які починаються з оберненого слеша `\` і використовуються для вставки спеціальних символів або керуючих послідовностей у рядки. Вони дають змогу включати символи, які в іншому випадку могли б призвести до спотворення рядка або змінити його зміст.

1. `\n` – перенесення рядка (новий рядок).
2. `\t` – табуляція (горизонтальна).
3. `\\` – обернений слеш.
4. `\'` – одинарна лапка.
5. `\"` – подвійна лапка.

```
print("Це перший рядок\nA це другий рядок")
```

Вивід:

```
Це перший рядок
A це другий рядок
```

Щоб уникнути конфліктів зі спеціальними символами в рядках, екранування може бути використане, щоб вставити ці символи в рядок без будь-яких непередбачуваних наслідків.

8.4. Неформатовані рядки

У Python неформатований рядок (raw string) – це рядок, у якому екрановані послідовності трактуються буквально, без будь-яких спроб інтерпретації їх як спеціальних символів. Це дає змогу вставляти обернений слеш `\` без необхідності екранування, що особливо корисно, наприклад, під час роботи з регулярними виразами або шляхами файлів у Windows.

Неформатовані рядки створюються за допомогою префікса `r` перед відкриваючими лапками. Ось кілька прикладів:

1. Рядок зі звичайним екрануванням:

```
path = "C:\\Users\\Username\\Documents"
print(path)  # Виведе: C:\Users\Username\Documents
```

2. Неформатований рядок:

```
path = r"C:\Users\Username\Documents"
print(path)  # Виведе: C:\Users\Username\Documents
```

В обох випадках виведення буде однаковим, але використання неформатованого рядка робить код більш зрозумілим і зручним, особливо коли маєте справу з багатоплутаними шляхами файлів або рядками, які містять багато обернених слешів.

8.5. Багаторядкові блоки тексту

У Python багаторядкові блоки тексту – це рядки, які займають кілька рядків і використовуються для документування коду, включення багаторядкових рядків коментарів або визначення багаторядкових рядків документації (docstring). Існують кілька способів створення багаторядкових блоків тексту в Python.

1. Три подвійні лапки (три одинарні лапки):

```
multiline_string = """
Це перший рядок.
Це другий рядок.
Це третій рядок.
"""
```

2. Три одинарні лапки (три подвійні лапки):

```
multiline_string = '''
Це перший рядок.
Це другий рядок.
Це третій рядок.
'''
```

3. Використання обернених слешів:

```
multiline_string = "Це перший рядок. \
Це другий рядок. \
Це третій рядок."
```

8.6. Доступ за індексами

У Python доступ до окремих символів у рядку здійснюється за допомогою індексів. Індекси в Python починаються з 0 для першого символу в рядку і можуть бути від'ємними для доступу до символів з кінця рядка.

```
my_string = "Python"

# Доступ до першого символу (P)
first_char = my_string[0]
print(first_char)

# Доступ до останнього символу (n)
last_char = my_string[-1]
print(last_char)

# Доступ до другого символу (y)
second_char = my_string[1]
print(second_char)

# Зміна символу в рядку
my_string = "Python"
my_string[0] = "J"  # Це призведе до помилки, оскільки рядки є не змінюваними
```

Якщо ви спробуєте змінити символ у рядку (як в останньому прикладі), ви отримаєте помилку, оскільки рядки є не змінюваними (*immutable*) в Python. Ви можете отримати значення символу, але не можете змінити його без створення нового рядка.

Також важливо пам'ятати, що доступ за індексами, що виходять за межі рядка, призведе до помилки. Наприклад, якщо рядок має довжину 6 символів і ви спробуєте звернутися до символу за індексом 6, ви отримаєте помилку "IndexError: string index out of range".

8.7. Зрізи

У Python зрізи (slices) дають змогу отримувати підрядок з рядка, вказуючи початковий та кінцевий індекси, а також крок (step), який указує крок переміщення між індексами.

```
my_string = "Python is awesome"

# Отримання підрядка "Py"
substring1 = my_string[0:2]
print(substring1)

# Отримання підрядка "is"
substring2 = my_string[7:9]
print(substring2)

# Отримання підрядка "awe"
substring3 = my_string[10:13]
print(substring3)

# Отримання підрядка "on is aw"
substring4 = my_string[4:13]
print(substring4)

# Використання кроку: отримання кожного другого символу
every_other_char = my_string[::2]
print(every_other_char)

# Використання від'ємних індексів: отримання рядка у зворотному порядку
reversed_string = my_string[::-1]
print(reversed_string)
```

У вищенаведених прикладах використовуються такі зрізи:

- *my_string[start:end]* – повертає підрядок, що починається з індексу *start* і закінчується перед індексом *end;*

- *my_string[start:end:step]* – повертає підрядок з кроком *step*, починаючи з індексу *start* і закінчуючи перед індексом *end.*

Якщо *start* або *end* не вказані, зріз відповідно починається з початку або закінчується на кінці рядка.

Якщо *step* не вказано, використовується значення за замовчуванням, яке дорівнює 1.

Важливо пам'ятати, що значення зрізу *[start:end*] включає символ за індексом start, але не включає символ за індексом *end.*

8.8. Отримання фрагменту рядка

Отримання фрагменту рядка в Python можна виконати за допомогою зрізів *(slices)*. Зрізи дають змогу Вам отримувати підрядки з вихідного рядка, вказуючи початковий та кінцевий індекси, а також крок переміщення між індексами, якщо потрібно.

1. Використання зрізу з початковим та кінцевим індексами:

```
my_string = "Python is awesome"
substring = my_string[7:9]  # Отримання підрядка "is"
print(substring)
```

2. Використання зрізу з початковим та кінцевим індексами разом з кроком:

```
my_string = "Python is awesome"
substring = my_string[4:13:2]  # Отримання кожного другого символу, починаючи з індексу 4
print(substring)
```

3. Використання від'ємних індексів для зрізу з кінця рядка:

```
my_string = "Python is awesome"
substring = my_string[-7:-1]  # Отримання підрядка "awesom"
print(substring)
```

4. Використання зрізу без початкового та кінцевого індексів:

```
my_string = "Python is awesome"
substring = my_string[:6]  # Отримання перших шести символів рядка
print(substring)
```

Зауважте, що в усіх цих прикладах отримані фрагменти рядка не включають кінцевий індекс. Тобто символ за кінцевим індексом не включається в результат.

8.9. Засоби перетворення рядків та одиночних символів. Функції int(), str(), repr(), float(), ord(), chr()

Функції перетворення дають змогу змінювати типи даних між рядками, числами та символами.

1. *int()* – перетворює рядок або число з плаваючою точкою в ціле число.

```
num_str = "123"
num_int = int(num_str)
print(num_int)  # Виведе: 123
```

2. *str()* – перетворює об’єкт у рядок.

```
num = 123
num_str = str(num)
print(num_str)  # Виведе: '123'
```

3. *repr()* – повертає рядкове представлення об’єкта, яке може бути використано для перевірки типу об’єкта.

```
num = 123
repr_str = repr(num)
print(repr_str)  # Виведе: '123'
```

4. *float()* – перетворює рядок або ціле число в число з плаваючою точкою.

```
num_str = "123.45"
num_float = float(num_str)
print(num_float)  # Виведе: 123.45
```

5. *ord()* – повертає ціле число, що відповідає кодовому значенню символу Unicode.

```
char = 'A'
unicode_val = ord(char)
print(unicode_val)  # Виведе: 65
```

6. *chr()* – повертає символ, що відповідає заданому кодовому значенню Unicode.

```
unicode_val = 65
char = chr(unicode_val)
print(char)  # Виведе: 'A'
```

8.10. Функції len(), max(), min()

1. *len()* – приймає рядок як аргумент і повертає його довжину, тобто кількість символів у рядку.

```
my_string = "Hello, world!"
length = len(my_string)
print(length)  # Виведе: 13
```

2. *max()* – приймає рядок як аргумент і повертає найбільший символ у рядку, визначений за їх кодовими значеннями Unicode.

```
my_string = "Hello, world!"
max_char = max(my_string)
print(max_char)  # Виведе: 'w'
```

3. *min()* – приймає рядок як аргумент і повертає найменший символ у рядку, визначений за їх кодовими значеннями Unicode.

```
my_string = "Hello, world!"
min_char = min(my_string)
print(min_char)  # Виведе: ' '
```

8.11. Доступ за індексами

Доступ до елементів рядка в Python здійснюється за допомогою індексів. Індексація в Python починається з нуля, тобто перший елемент має індекс 0, другий – 1, і так далі. Для отримання доступу до елементу за його індексом використовуються квадратні дужки `[]`.

1. Позитивні індекси: Індекси можуть бути позитивними числами, які вказують на позицію елемента від початку рядка, починаючи з 0.

```
my_string = "Hello"
print(my_string[0])  # Виведе: 'H'
print(my_string[1])  # Виведе: 'e'
print(my_string[4])  # Виведе: 'o'
```

2. Від'ємні індекси: Також можна використовувати від'ємні індекси, які вказують на позицію елемента від кінця рядка, починаючи з -1.

```
my_string = "Hello"
print(my_string[-1])  # Виведе: 'o'
print(my_string[-2])  # Виведе: 'l'
```

3. Доступ до підрядка: Також можна використовувати зрізи для отримання підрядка рядка.

```
my_string = "Hello, World!"
print(my_string[7:12])  # Виведе: 'World'
```

Зверніть увагу, що доступ до елемента за індексом, що виходить за межі рядка, призведе до помилки. Щоб уникнути цього, перевірте довжину рядка перед доступом за індексом.

Перелік питань для самоконтролю

1. Рядковий тип даних.
2. Операції над рядками.
3. Неформатовані рядки.
4. Багаторядкові блоки тексту.
5. Зрізи.
6. Засоби перетворення рядків та одиночних символів. Функції ***int(), str(), repr(), float(), ord(), chr().***
7. Функції ***len(), max(), min().***

Тестові питання

1. Який метод використовується для об'єднання (конкатенації) двох рядків у Python:

а) `concat()`;
б) `join()`;
в) `merge()`;
г) `+` (плюс).

2. Який оператор дає змогу вибрати підстроку з рядка в Python:

а) `substring()`;
б) `slice[]`;
в) `sub()`;
г) `substr()`.

3. Яка функція використовується для знаходження довжини рядка в Python:

а) `length()`;
б) `len()`;
в) `size()`;
г) `count()`.

4. Яка функція використовується для заміни певних символів у рядку на інші:

а) `replace()`;
б) `sub()`;
в) `swap()`;
г) `switch()`.

5. Який метод дає змогу перетворити рядок на рядок з великими літерами в Python:

а) `to_upper()`;
б) `upper()`;
в) `uppercase()`;
г) `toUpper()`.

6. Який метод використовується для розбиття рядка на список підстрок за певним роздільником у Python:

а) `split()`;
б) `divide()`;
в) `break()`;
г) `explode()`.

7. Яка функція використовується для знаходження позиції підрядка у рядку в Python:

а) `index()`;
б) `position()`;
в) `find()`;
г) `locate()`.

8. Який оператор дає змогу перевірити, чи містить один рядок підстроку в Python:

а) `contains`;

б) `in`;

в) `exists()`;

г) `has()`.

9. Яка функція використовується для видалення пробільних символів з початку та кінця рядка в Python:

а) `trim()`;

б) `strip()`;

в) `remove_whitespace()`;

г) `clear()`.

10. Який метод дає змогу перетворити рядок на рядок з малими літерами в Python:

а) `to_lower()`;

б) `lower()`;

в) `lowercase()`;

г) `toLower()`.

Рекомендовані джерела:

Основні: 1; 2, 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть програму, яка перевіряє, чи є введене користувачем слово паліндромом (слово, яке читається однаково зліва направо і справа наліво).

2. Напишіть програму, яка запитує в користувача рядок і виводить його у верхньому та нижньому регістрі.

3. Створіть програму, яка приймає рядок від користувача й підраховує кількість слів у цьому рядку. Слова вважаються розділеними пробілами.

4. Напишіть програму, яка видаляє всі пробіли з рядка, введеного користувачем.

5. Створіть програму, яка перевіряє, чи містить введений рядок певний підрядок. Підрядок також задається користувачем.

6. Напишіть програму, яка перевертає введений рядок, тобто виводить рядок у зворотному порядку.

7. Напишіть програму, яка замінює всі входження одного підрядка на інший у рядку, введеному користувачем.

8. Створіть програму, яка підраховує кількість разів появи певного символу в рядку. Символ і рядок задаються користувачем.

9. Напишіть програму, яка форматуватиме рядок, використовуючи f-рядки для вставки змінних у рядок. Задайте користувачу три змінні і виведіть їх у форматованому рядку.

10. Напишіть програму, яка обрізає пробіли з початку та кінця рядка, введеного користувачем, використовуючи метод `strip()`.

Тема 9
Функції роботи з рядками

9.1. Функції для роботи з рядками, що визначають особливості рядка: isalnum(), isalpha(), isascii(), isdecimal(), isdigit(), isidentifier(), islower(), isnumeric(), isprintable(), isspace(), istitle(), isupper().

9.2. Функції пошуку та заміни підрядка в рядку: count(), find(), index(), rfind(), rindex(), replace().

9.3. Функції, що визначають та обробляють початок та кінець рядка: endswith(), startswith(), lstrip(), rstrip(), strip().

9.4. Функції обробки рядка згідно з форматом чи правилом кодування: encode(), expandtabs(), format(). Стилі форматування.

9.5. Функції вирівнювання рядків: center(), ljust(), rjust(), zfill().

9.6. Функції, які змінюють регістр символів у рядку: capitalize(), casefold(), lower(), swapcase(), title(), upper().

9.7. Функції розбиття рядків на частини та утворення нових рядків за допомогою кортежів та списків: join(), partition(), rpartition(), rsplit(), split(), splitlines().

9.8. Операції форматування рядків. Вирази форматування рядків. Бінарний оператор %.

9.9. F-рядок.

9.1. Функції для роботи з рядками, що визначають особливості рядка: isalnum(), isalpha(), isascii(), isdecimal(), isdigit(), isidentifier(), islower(), isnumeric(), isprintable(), isspace(), istitle(), isupper()

1. *isalnum()* – повертає *True,* якщо всі символи в рядку є алфавітно-цифровими (літерами або цифрами), інакше повертає *False.*

```
my_string = "Hello123"
print(my_string.isalnum())  # Виведе: True
```

2. *isalpha()* – повертає *True*, якщо всі символи в рядку є літерами, інакше повертає *False.*

```
my_string = "Hello"
print(my_string.isalpha())  # Виведе: True
```

3. *isascii()* – повертає *True,* якщо всі символи в рядку є символами ASCII, інакше повертає *False.*

```
my_string = "Hello"
print(my_string.isascii())  # Виведе: True
```

4. *isdecimal()* – повертає *True,* якщо всі символи в рядку є десятковими цифрами, інакше повертає *False.*

```
my_string = "123"
print(my_string.isdecimal())  # Виведе: True
```

5. *isdigit()* – повертає *True*, якщо всі символи в рядку є цифрами, інакше повертає *False.*

```
my_string = "123"
print(my_string.isdigit())  # Виведе: True
```

6. *isidentifier()* – повертає *True,* якщо рядок є допустимим ідентифікатором Python, інакше повертає *False.*

```
my_string = "hello_world"
print(my_string.isidentifier())  # Виведе: True
```

7. *islower()* – повертає *True*, якщо всі букви в рядку є малими (нижній регістр), інакше повертає *False.*

```
my_string = "hello"
print(my_string.islower())  # Виведе: True
```

8. *isnumeric()* – повертає *True*, якщо всі символи в рядку є числовими, інакше повертає *False.*

```
my_string = "123"
print(my_string.isnumeric())  # Виведе: True
```

9. *isprintable()* – повертає *True,* якщо всі символи в рядку є друкованими (тобто можуть бути виведені на екран), інакше повертає *False.*

```
my_string = "Hello"
print(my_string.isprintable())  # Виведе: True
```

10. *isspace()* – повертає *True*, якщо всі символи в рядку є пробілами, інакше повертає *False.*

```
my_string = "   "
print(my_string.isspace())  # Виведе: True
```

11. *istitle()* – повертає *True,* якщо рядок починається з великої літери, і всі інші символи в рядку є малими, інакше повертає *False.*

```
my_string = "Hello World"
print(my_string.istitle())  # Виведе: True
```

12. *isupper()* – повертає *True*, якщо всі букви в рядку є великими (верхній регістр), інакше повертає *False.*

```
my_string = "HELLO"
print(my_string.isupper())  # Виведе: True
```

Ці функції дають змогу Вам виконувати різноманітні перевірки та визначення різних характеристик рядків у Python.

9.2. Функції пошуку та заміни підрядка в рядку: count(), find(), index(), rfind(), rindex(), replace()

1. *count(substring[, start[, end]])* – повертає кількість входжень підрядка *substring* у рядок. Може приймати параметри *start* і *end,* що вказують на початковий та кінцевий індекси для пошуку.

```
my_string = "hello world"
print(my_string.count("o"))  # Виведе: 2
```

2. *find(substring[, start[, end]])* – повертає індекс першого входження підрядка *substring* у рядок. Якщо підрядок не знайдено, повертає -1. Може приймати параметри *start і end,* що вказують на початковий та кінцевий індекси для пошуку.

```
my_string = "hello world"
print(my_string.find("o"))  # Виведе: 4
```

3. *index(substring[, start[, end]])* – аналогічно *find()*, повертає індекс першого входження підрядка *substring* у рядок. Проте, якщо підрядок не знайдено, генерує виняток *ValueError.* Може приймати параметри *start* і *end*, що вказують на початковий та кінцевий індекси для пошуку.

```
my_string = "hello world"
print(my_string.index("o"))  # Виведе: 4
```

4. *rfind(substring[, start[, end]])* – повертає індекс останнього входження підрядка *substring* у рядок. Якщо підрядок не знайдено, повертає -1. Може приймати параметри *start* і *end*, що вказують на початковий та кінцевий індекси для пошуку.

```
my_string = "hello world"
print(my_string.rfind("o"))  # Виведе: 7
```

5. *rindex(substring[, start[, end]])* – аналогічно *rfind()*, повертає індекс останнього входження підрядка *substring* у рядок. Проте, якщо підрядок не знайдено, генерує виняток *ValueError*. Може приймати параметри *start* і *end*, що вказують на початковий та кінцевий індекси для пошуку.

```
my_string = "hello world"
print(my_string.rindex("o"))  # Виведе: 7
```

6. *replace(old, new[, count])* – замінює всі входження підрядка *old* на підрядок *new* у рядку. Приймає опціональний параметр *count*, що вказує на максимальну кількість замін, яку потрібно зробити.

```
my_string = "hello world"
new_string = my_string.replace("o", "0")
print(new_string)  # Виведе: hell0 w0rld
```

9.3. Функції, що визначають та обробляють початок та кінець рядка: endswith(), startswith(), lstrip(), rstrip(), strip()

1. *endswith(suffix[, start[, end]])* – повертає *True*, якщо рядок закінчується підрядком *suffix*, інакше повертає *False*. Може приймати параметри *start* і *end*, що вказують на початковий та кінцевий індекси для перевірки.

```
my_string = "hello world"
print(my_string.endswith("world"))  # Виведе: True
```

2. *startswith(prefix[, start[, end]])* – повертає *True*, якщо рядок починається з підрядка *prefix*, інакше повертає *False*. Може приймати параметри *start* і *end*, що вказують на початковий та кінцевий індекси для перевірки.

```
my_string = "hello world"
print(my_string.startswith("hello"))  # Виведе: True
```

3. *lstrip([characters])* – повертає копію рядка, з якого видалено всі символи зліва, що зустрічаються в параметрі *characters* (за замовчуванням – пробіли).

```
my_string = "   hello   "
new_string = my_string.lstrip()
print(new_string)  # Виведе: "hello   "
```

4. *rstrip([characters])* – повертає копію рядка, з якого видалено всі символи справа, що зустрічаються в параметрі *characters* (за замовчуванням - пробіли).

```
my_string = "   hello   "
new_string = my_string.rstrip()
print(new_string)  # Виведе: "   hello"
```

5. *strip([characters])* – повертає копію рядка, з якого видалено всі символи зліва та справа, що зустрічаються в параметрі *characters* (за замовчуванням - пробіли).

```
my_string = "   hello   "
new_string = my_string.strip()
print(new_string)  # Виведе: "hello"
```

Ці функції дають змогу Вам визначати й обробляти префікси та суфікси рядка, а також видаляти зайві пробіли з початку й кінця рядка.

9.4. Функції обробки рядка згідно з форматом чи правилом кодування: encode(), expandtabs(), format(). Стилі форматування

1. *encode([encoding='utf-8', errors='strict'])* – повертає байтовий об'єкт, який являє собою рядок, закодований за вказаним кодуванням *encoding*. Параметр *errors* вказує, як обробляти помилки кодування.

```
my_string = "Привіт"
encoded_string = my_string.encode(encoding='utf-8')
print(encoded_string)
# Виведе: b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd1\x96\xd1\x82'
```

2. *expandtabs([tabsize=8])* – повертає копію рядка, де всі символи табуляції (`\t`) замінені на пробіли відповідно до розміру табуляції *tabsize*.

```
my_string = "hello\tworld"
expanded_string = my_string.expandtabs(tabsize=4)
print(expanded_string)  # Виведе: "hello   world"
```

3. *format(*args, kwargs)* – форматує рядок, використовуючи передані аргументи *args* та ключові аргументи *kwargs*. Дає змогу вставляти значення у визначені місця в рядку за допомогою фігурних дужок {}.

```
name = "John"
age = 30
formatted_string = "My name is {} and I am {} years old".format(name, age)
print(formatted_string)  # Виведе: "My name is John and I am 30 years old"
```

4. Стилі форматування.

1. Старий спосіб:

```
name = "John"
age = 30
formatted_string = "My name is %s and I am %d years old" % (name, age)
print(formatted_string)  # Виведе: "My name is John and I am 30 years old"
```

2. f-рядки (з Python 3.6+):

```
name = "John"
age = 30
formatted_string = f"My name is {name} and I am {age} years old"
print(formatted_string)  # Виведе: "My name is John and I am 30 years old"
```

3. Метод `str.format()` (з Python 2.7+):

```
name = "John"
age = 30
formatted_string = "My name is {} and I am {} years old".format(name, age)
print(formatted_string)  # Виведе: "My name is John and I am 30 years old"
```

Ці функції дають змогу Вам працювати з рядками відповідно до вказаних форматів чи правил кодування, а також забезпечують різноманітні способи форматування рядків у Python.

9.5. Функції вирівнювання рядків: center(), ljust(), rjust(), zfill()

1. *center(width[, fillchar])* – повертає копію рядка, вирівняну по центру до вказаної ширини *width*. Параметр *fillchar* вказує символ, який буде використаний для заповнення відсутніх символів. За замовчуванням – пробіл.

```
my_string = "hello"
centered_string = my_string.center(10, "*")
print(centered_string)  # Виведе: "hello*"
```

2. *ljust(width[, fillchar])* – повертає копію рядка, вирівняну по лівому краю до вказаної ширини *width*. Параметр *fillchar* вказує символ, який буде використаний для заповнення відсутніх символів. За замовчуванням – пробіл.

```
my_string = "hello"
left_aligned_string = my_string.ljust(10, "*")
print(left_aligned_string)  # Виведе: "hello*"
```

3. *rjust(width[, fillchar])* – повертає копію рядка, вирівняну по правому краю до вказаної ширини *width*. Параметр *fillchar* вказує символ, який буде використаний для заповнення відсутніх символів. За замовчуванням – пробіл.

```
my_string = "hello"
right_aligned_string = my_string.rjust(10, "*")
print(right_aligned_string)  # Виведе: "*hello"
```

4. *zfill(width)* – повертає копію рядка, вирівняну по правому краю до вказаної ширини *width*, доповнюючи рядок нулями зліва. Використовується часто для чисел.

```
my_string = "42"
zero_filled_string = my_string.zfill(5)
print(zero_filled_string)  # Виведе: "00042"
```

9.6. Функції, які змінюють регістр символів у рядку: capitalize(), casefold(), lower(), swapcase(), title(), upper()

1. *capitalize()* – повертає копію рядка, в якій перший символ перетворений на верхній регістр, а всі інші – на нижній.

```
my_string = "hello world"
capitalized_string = my_string.capitalize()
print(capitalized_string)  # Виведе: "Hello world"
```

2. *casefold()* – повертає копію рядка, в якій усі символи перетворені на нижній регістр, включаючи символи, що мають відповідний нижній регістр у Unicode. Цей метод корисний для порівняння рядків без урахування регістру і для рядків, що містять нестандартні символи.

```
my_string = "Hello World"
casefolded_string = my_string.casefold()
print(casefolded_string)  # Виведе: "hello world"
```

3. *lower()* – повертає копію рядка, в якій усі символи перетворені на нижній регістр.

```
my_string = "HELLO WORLD"
lowercased_string = my_string.lower()
print(lowercased_string)  # Виведе: "hello world"
```

4. *swapcase()* – повертає копію рядка, в якій верхні символи замінені на нижні, а нижні – на верхні.

```
my_string = "Hello World"
swapped_string = my_string.swapcase()
print(swapped_string)  # Виведе: "hELLO wORLD"
```

5. *title()* – повертає копію рядка, в якій перший символ кожного слова перетворений на верхній регістр, а всі інші – на нижній.

```
my_string = "hello world"
titled_string = my_string.title()
print(titled_string)  # Виведе: "Hello World"
```

6. *upper()* – повертає копію рядка, в якій усі символи перетворені на верхній регістр.

```
my_string = "hello world"
uppercased_string = my_string.upper()
print(uppercased_string)  # Виведе: "HELLO WORLD"
```

9.7. Функції розбиття рядків на частини та утворення нових рядків за допомогою кортежів та списків: join(), partition(), rpartition(), rsplit(), split(), splitlines()

1. *join(iterable)* – повертає рядок, який утворюється з об'єднанням рядків зі списку (або іншого ітерабельного об'єкта), розділених роздільником, що визначений методом, до якого він був викликаний.

```
my_list = ["Hello", "world"]
joined_string = " ".join(my_list)
print(joined_string)  # Виведе: "Hello world"
```

2. *partition(sep)* – розбиває рядок на три частини: все до першого входження підрядка *sep*, сам підрядок *sep* і все після нього. Повертає кортеж з трьома елементами.

```
my_string = "hello world"
partitioned_string = my_string.partition(" ")
print(partitioned_string)  # Виведе: ('hello', ' ', 'world')
```

3. *rpartition(sep)* – розбиває рядок на три частини: все до останнього входження підрядка *sep*, сам підрядок *sep* і все після нього. Повертає кортеж з трьома елементами.

```
my_string = "hello world hello"
partitioned_string = my_string.rpartition(" ")
print(partitioned_string)  # Виведе: ('hello world', ' ', 'hello')
```

4. *rsplit(sep=None, maxsplit=-1)* – розбиває рядок на список рядків, використовуючи *sep* як роздільник, починаючи з кінця рядка. Максимальна кількість розбиттів задається параметром *maxsplit*. Повертає список рядків.

```
my_string = "hello world"
split_string = my_string.rsplit(" ")
print(split_string)  # Виведе: ['hello', 'world']
```

5. *split(sep=None, maxsplit=-1)* – розбиває рядок на список рядків, використовуючи *sep* як роздільник. Максимальна кількість розбиттів задається параметром *maxsplit*. Повертає список рядків.

```
my_string = "hello world"
split_string = my_string.split(" ")
print(split_string)  # Виведе: ['hello', 'world']
```

6. *splitlines(keepends=False)* – розбиває рядок на список рядків по символах нового рядка (`\n`, `\r`, або `\r\n`). Параметр *keepends* вказує, чи потрібно зберігати символи нового рядка в кожному рядку. Повертає список рядків.

```
my_string = "hello\nworld"
split_lines = my_string.splitlines()
print(split_lines)  # Виведе: ['hello', 'world']
```

9.8. Операції форматування рядків. Вирази форматування рядків. Бінарний оператор %

Операції форматування рядків у Python використовуються для вставки значень у рядок за допомогою спеціального синтаксису. Основні методи форматування рядків у Python охоплюють використання оператора `%` та методу *format()*.

1. Вирази форматування рядків з використанням бінарного оператора `%`:

- `%s` – вставляє рядкове представлення об’єкта;
- `%d` – вставляє ціле число;
- `%f` – вставляє десяткове число;
- `%x`, `%X` – вставляє шістнадцяткове представлення числа (нижній або верхній регістр);
- `%o` – вставляє вісімкове представлення числа;
- `%r` – вставляє рядок, використовуючи `repr()`.

Синтаксис: "%формат" % (змінна1, змінна2, ...)

```
name = "John"
age = 30
formatted_string = "My name is %s and I am %d years old" % (name, age)
print(formatted_string)  # Виведе: "My name is John and I am 30 years old"
```

2. Метод *format():*

Синтаксис: "{}".format(змінна1, змінна2, ...)

```
name = "John"
age = 30
formatted_string = "My name is {} and I am {} years old".format(name, age)
print(formatted_string)  # Виведе: "My name is John and I am 30 years old"
```

3. f-рядки (з Python 3.6+):

Синтаксис: f"{'змінна1'} {'змінна2'}"

```
name = "John"
age = 30
formatted_string = f"My name is {name} and I am {age} years old"
print(formatted_string)  # Виведе: "My name is John and I am 30 years old"
```

9.9. F-рядок

F-рядки (або f-рядки) – це спеціальний механізм форматування рядків, який був доданий у Python 3.6. Вони дають змогу вбудовувати значення змінних безпосередньо в рядок, використовуючи синтаксис, подібний до виразів Python.

1. Синтаксис:

F-рядки відрізняються від звичайних рядків тим, що вони мають префікс *f* або *F*, після якого йде рядок з фігурними дужками {} для вставки значень змінних.

```
variable = "value"
f_string = f"This is an {variable}."
```

2. Використання змінних:

Значення змінних вставляються безпосередньо в рядок за допомогою фігурних дужок {}.

```
name = "John"
age = 30
f_string = f"My name is {name} and I am {age} years old."
```

3. Вирази та функції:

F-рядки також підтримують виконання виразів та виклики функцій.

```
num1 = 10
num2 = 20
f_string = f"The sum of {num1} and {num2} is {num1 + num2}."
```

4. Підтримка форматування:

F-рядки можуть також використовувати різні специфікатори форматування для змінних.

```
pi = 3.14159
f_string = f"The value of pi is {pi:.2f}."
```

5. Багаторядкові рядки:

F-рядки також можуть бути багаторядковими, використовуючи потрійні лапки `"""` або `""`.

```
name = "John"
f_string = f"""
Hello,
My name is {name}.
Nice to meet you!
"""
```

F-рядки надають зручний та зрозумілий спосіб вставки значень змінних у рядки, що робить код читабельнішим та ефективнішим.

Перелік питань для самоконтролю

1. Функції: **isalnum(), isalpha(), isascii(), isdecimal(), isdigit(), isidentifier(), islower(), isnumeric(), isprintable(), isspace(), istitle(), isupper().**
2. Функції: **count(), find(), index(), rfind(), rindex(), replace().**
3. Функції: **endswith(), startswith(), lstrip(), rstrip(), strip().**
4. Функції обробки рядка: **encode(), expandtabs(), format()**. Стилі форматування.
5. Функції вирівнювання рядків: **center(), ljust(), rjust(), zfill().**
6. Функції: **capitalize(), casefold(), lower(), swapcase(), title(), upper().**

Тестові питання

1. Яка функція використовується для визначення довжини рядка в Python:

а) `size()`;

б) `length()`;

в) `count()`;

г) `len()`.

2. Яка функція використовується для перетворення рядка у верхній регістр (великі літери) в Python:

а) `upper()`;

б) `uppercase()`;

в) `to_upper()`;

г) `capitalize()`.

3. Яка функція використовується для перетворення рядка в нижній регістр (малі літери) в Python:

а) `to_lower()`;

б) `lowercase()`;

в) `lower()`;

г) `case_lower()`.

4. Яка функція використовується для перевірки того, чи починається рядок з певного підрядка в Python:

а) `start_with()`;

б) `begin_with()`;

в) `startswith()`;

г) `start()`.

5. Яка функція використовується для заміни підрядка в рядку на іншу підстроку в Python:

а) `replace()`;

б) `substitute()`;

в) `swap()`;

г) `change()`.

6. Яка функція використовується для розділення рядка на список підстрок за певним роздільником у Python:

а) `split()`;

б) `break()`;

в) `divide()`;

г) `slice()`.

7. Яка функція використовується для видалення пробілів на початку та в кінці рядка в Python:

а) `trim()`;

б) `strip()`;

в) `clean()`;

г) `clear()`.

8. Яка функція використовується для з'єднання рядків у один рядок в Python:

а) `merge()`;

б) `join()`;

в) `concatenate()`;

г) `combine()`.

9. Яка функція використовується для перевірки того, чи містить рядок певну підстроку в Python:

а) `contains()`;

б) `include()`;

в) `has()`;

г) `in()`.

10. Яка функція використовується для видалення певного символу з рядка в Python:

а) `delete()`;

б) `remove()`;

в) `erase()`;

г) `replace()`.

Рекомендовані джерела:

Основні: 1; 2; 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть програму, яка запитує в користувача ім’я та вік, а потім виводить повідомлення з використанням f-рядків.

2. Створіть програму, яка запитує в користувача число з плаваючою комою і виводить його з точністю до трьох десяткових знаків за допомогою f-рядків.

3. Напишіть програму, яка запитує в користувача два рядки та виводить їх у форматі таблиці з вирівнюванням по центру за допомогою методу `str.format()`.

4. Створіть програму, яка запитує в користувача день, місяць і рік, а потім виводить дату у форматі "ДД-ММ-РР" за допомогою методу `str.format()`.

5. Напишіть програму, яка запитує в користувача число та ширину поля і виводить число з указаною шириною поля, вирівняне по правому краю за допомогою f-рядків.

6. Створіть програму, яка запитує в користувача число і ширину поля, а потім виводить число з заповнювачем нулями з використанням методу `str.format()`.

7. Напишіть програму, яка запитує в користувача ім’я, вік і зріст, а потім виводить ці дані у форматованому рядку за допомогою f-рядків.

8. Створіть програму, яка запитує в користувача число і виводить його в науковому форматі за допомогою методу `str.format()`.

9. Напишіть програму, яка запитує в користувача три рядки і виводить їх у вигляді таблиці з трьома стовпчиками за допомогою f-рядків.

10. Створіть програму, яка запитує в користувача ім’я, вік і рейтинг, а потім виводить ці дані у форматованому рядку з використанням методу `str.format()`.

11. Напишіть програму, яка використовує бінарний оператор `%` для форматування рядка, що включає ім’я та вік користувача.

12. Створіть програму, яка використовує оператор `%` для форматування чисел з плаваючою комою з точністю до двох десяткових знаків.

13. Напишіть програму, яка використовує оператор `%` для форматування рядків з вирівнюванням по правому краю.

14. Створіть програму, яка використовує оператор `%` для форматування чисел з указанням мінімальної ширини поля.

15. Напишіть програму, яка використовує оператор `%` для форматування дати у вигляді "день-місяць-рік".

16. Створіть програму, яка використовує оператор `%` для форматування кількох змінних у одному рядку.

17. Напишіть програму, яка використовує оператор `%` для форматування чисел у науковому форматі.

18. Створіть програму, яка використовує оператор `%` для форматування рядків у вигляді таблиці з вирівнюванням по центру.

19. Напишіть програму, яка використовує оператор `%` для форматування рядків, де заповнювачем є нулі.

20. Створіть програму, яка використовує оператор `%` для форматування рядків з різними типами даних, такими як рядок, число та дата.

21. Напишіть функцію `check_alnum(text)`, яка перевіряє, чи є рядок `text` алфанумеричним (містить лише літери та цифри).

22. Створіть функцію `check_alpha(text)`, яка перевіряє, чи містить рядок `text` тільки літери.

23. Напишіть функцію `check_ascii(text)`, яка перевіряє, чи містить рядок `text` тільки ASCII символи.

24. Створіть функцію `check_decimal(text)`, яка перевіряє, чи містить рядок `text` тільки десяткові цифри.

25. Напишіть функцію `check_digit(text)`, яка перевіряє, чи містить рядок `text` тільки цифри.

26. Створіть функцію `check_identifier(text)`, яка перевіряє, чи є рядок `text` дійсним ідентифікатором змінної в Python.

27. Напишіть функцію `check_lower(text)`, яка перевіряє, чи є всі символи в рядку `text` у нижньому регістрі.

28. Створіть функцію `check_numeric(text)`, яка перевіряє, чи містить рядок `text` лише числові символи, охоплюючи числа, представлені у вигляді римських цифр, дробів і т. д.

29. Напишіть функцію `check_printable(text)`, яка перевіряє, чи є всі символи в рядку `text` друкованими.

30. Створіть функцію `check_space(text)`, яка перевіряє, чи містить рядок `text` тільки пробільні символи.

31. Напишіть функцію `check_title(text)`, яка перевіряє, чи є рядок `text` у заголовному регістрі (перше слово з великої літери).

32. Створіть функцію `check_upper(text)`, яка перевіряє, чи є всі символи в рядку `text` у верхньому регістрі.

Тема 10
Списки в мові Python

10.1. Означення, властивості та застосування списків у мові Python.

10.2. Визначення списку як змінюваної послідовності даних. Способи створення списку. Способи створення копії списку.

10.3. Операції над списками.

10.4. Особливості формування та застосування багатовимірних списків.

10.5. Перебір елементів списку.

10.6. Методи додавання та видалення елементів списку.

10.7. Пошук елемента списку й одержання відомостей про значення, які входять у список.

10.8. Визначення кількості елементів.

10.9. Перевертання та перемішування списку.

10.10. Вибір елементів списку випадковим чином за допомогою функцій модуля random.

10.11. Сортування списку.

10.12. Заповнення списку числами.

10.13. Перетворення списку на рядок.

10.14. Функція rang() та її параметри.

10.15. Генератори списків та вирази-генератори.

10.16. Властивості та параметри методу index().

10.17. Визначення параметрів елементів списку.

10.1. Означення, властивості та застосування списків у мові Python

Список у мові програмування Python є одним з найбільш універсальних та корисних типів даних.

1. Означення. Список у Python – це змінна довжина, яка дає змогу зберігати послідовність елементів у впорядкованій структурі даних. Елементи списку можуть бути будь-якого типу та їх можна змінювати (мутабельність).

2. Властивості списків:

1. Списки в Python є змінюваними, тобто їх можна змінювати після створення.

2. Елементи списку можуть бути будь-якого типу даних: числа, рядки, списки, кортежі, навіть функції та об’єкти.

3. Доступ до елементів списку відбувається за допомогою індексів.

4. Списки можуть містити дубльовані елементи.

5. Списки можуть бути вкладеними, тобто елементами списку можуть бути інші списки.

3. Застосування списків:

1. Зберігання послідовності даних: списки використовуються для зберігання послідовностей даних, таких як числа, рядки, об’єкти тощо.

2. Обробка та маніпуляція даними: списки дають змогу виконувати різноманітні операції з даними, такі як сортування, фільтрація, видалення елементів, додавання нових елементів тощо.

3. Передача даних функціям: списки можуть бути передані як аргументи функціям, що дає змогу створювати більш гнучкі та універсальні функції.

4. Робота зі структурованими даними: списки дають змогу створювати структуровані дані, такі як таблиці або бази даних, які можна легко обробляти та аналізувати.

Списки в Python є потужним інструментом для роботи з даними й дають змогу ефективно вирішувати широкий спектр завдань програмування. Вони є основним елементом багатьох програм і додатків у Python.

10.2. Визначення списку як змінюваної послідовності даних. Способи створення списку. Способи створення копії списку

Список у мові програмування Python є змінюваною послідовністю даних, що дає змогу зберігати колекцію елементів у впорядкованій формі.

1. Визначення списку як змінюваної послідовності даних. У Python список являє собою колекцію елементів, що можуть бути будь-якого типу даних і можуть бути змінені після створення. Елементи списку впорядковані, тобто зберігаються в тому самому порядку, у якому були додані.

2. Способи створення списку:

1. За допомогою квадратних дужок []:

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

2. За допомогою функції *list():*

```
my_list = list(range(1, 6))
```

3. За допомогою спискового виразу:

```
my_list = [x for x in range(1, 6)]
```

3. Способи створення копії списку:

1. Копія за допомогою зрізу `[:]`:

```
original_list = [1, 2, 3]
copy_list = original_list[:]
```

2. За допомогою методу *copy():*

```
original_list = [1, 2, 3]
copy_list = original_list.copy()
```

3. За допомогою функції *list():*

```
original_list = [1, 2, 3]
copy_list = list(original_list)
```

4. За допомогою спискового виразу:

```
original_list = [1, 2, 3]
copy_list = [x for x in original_list]
```

Ці способи дають змогу створити копію списку, що є корисним у випадках, коли потрібно працювати з оригінальним списком, не змінюючи його. Це важливо для уникнення непередбачених змін у даних.

10.3. Операції над списками

Операції над списками в мові програмування Python охоплюють різноманітні дії зі списками, такі як додавання елементів, видалення, зміна, сортування тощо.

1. Додавання елементів:

- додавання елемента в кінець списку *list.append(element)*;
- додавання елемента на певну позицію *list.insert(index, element)*;
- додавання всіх елементів з іншого списку *list.extend(other_list)*.

2. Видалення елементів:

- видалення першого елемента зі значенням `x` – *list.remove(x)*;
- видалення елемента за певним індексом – *del list[index]*;
- видалення та повернення елемента за певним індексом – *element = list.pop(index)*;
- видалення останнього елемента – *element = list.pop()*;
- видалення всіх елементів – *list.clear()*.

3. Зміна елементів. Зміна значення елемента за певним індексом – *list[index] = new_value*.

4. Сортування:

- сортування списку за зростанням – *list.sort()*;
- сортування списку за спаданням – *list.sort(reverse=true)*
- створення відсортованої копії списку – *sorted_list = sorted(list)*.

5. Інші операції:

- знаходження довжини списку – *length = len(list)*;
- перевірка наявності елемента у списку – *if element in list:* ;
- обхід елементів списку за допомогою циклу *for*: .

```
for element in list:
  print(element)
```

6. Пошук елементів. Знаходження індексу першого входження елемента зі значенням `x` - *index = list.index(x)*.

10.4. Особливості формування та застосування багатовимірних списків

Багатовимірні списки (також відомі як списки списків) є потужним інструментом для роботи зі складними структурами даних у мові програмування Python.

1. Формування багатовимірних списків. Багатовимірні списки можна створити за допомогою вкладення одного списку в інший.

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

2. Особливості індексації. Доступ до елементів багатовимірного списку відбувається за допомогою подвійної індексації. Наприклад, для доступу до елементу в рядку `i` та стовпчику `j` використовується синтаксис *matrix[i][j]*.

3. Застосування багатовимірних списків:

1. Матриці та таблиці. Багатовимірні списки зручно використовувати для зберігання та обробки матриць, таблиць та інших структур даних, що мають двовимірну форму.

2. Ігрове програмування. В ігровому програмуванні багатовимірні списки можна використовувати для зберігання стану гри, розташування об’єктів на полі тощо.

3. Обробка зображень. В обробці зображень багатовимірні списки можуть використовуватися для зберігання пікселів зображення в форматі RGB або інших.

4. Маніпуляції з багатовимірними списками:

1. Додавання рядків або стовпців. Додавання нових рядків або стовпців до багатовимірного списку.

2. Видалення рядків або стовпців. Видалення існуючих рядків або стовпців.

3. Зміна елементів. Зміна значень елементів у різних рядках та стовпцях.

4. Обхід. Обхід усіх елементів багатовимірного списку за допомогою вкладених циклів.

Багатовимірні списки дають змогу представляти та маніпулювати багатовимірними даними у Python зручним та ефективним способом. Вони є важливою частиною багатьох програм, що вимагають обробки складних структур даних.

10.5. Перебір елементів списку

Перебір елементів списку в Python можна виконати за допомогою циклу *for* або з використанням індексів.

1. За допомогою циклу *for*. Використання циклу *for* є найбільш зручним способом для перебору елементів списку:

```
my_list = [1, 2, 3, 4, 5]
for element in my_list:
    print(element)
```

Цей код виведе кожен елемент списку *my_list* на новому рядку.

2. З використанням індексів:

Іноді потрібно звертатися до елементів списку за їх індексами або потрібно знати індекс кожного елемента. Для цього можна використовувати вбудовану функцію *range()* в поєднанні з функцією *len()* для отримання довжини списку:

```
my_list = [1, 2, 3, 4, 5]
for i in range(len(my_list)):
    print(my_list[i])
```

Цей код також виведе кожен елемент списку *my_list* на новому рядку, але використовує індекси для доступу до елементів.

Обидва ці методи є корисними залежно від конкретного завдання. Якщо Вам просто потрібно перебрати елементи списку, то краще використовувати цикл *for*, а якщо Вам потрібен доступ до індексів елементів, ви можете використовувати індекси.

10.6. Методи додавання та видалення елементів списку

У мові програмування Python існують різні методи для додавання та видалення елементів у списку.

1. Методи додавання елементів:

- *append()* – цей метод додає елемент у кінець списку.

```
my_list = [1, 2, 3]
my_list.append(4)
# my_list тепер буде [1, 2, 3, 4]
```

- *insert()* – цей метод додає елемент на певну позицію у списку.

```
my_list = [1, 2, 3]
my_list.insert(1, 5)  # Додає 5 на позицію з індексом 1
# my_list тепер буде [1, 5, 2, 3]
```

- *extend()* – цей метод додає всі елементи іншого списку в кінець поточного списку.

```
my_list = [1, 2, 3]
other_list = [4, 5, 6]
my_list.extend(other_list)
# my_list тепер буде [1, 2, 3, 4, 5, 6]
```

2. Методи видалення елементів:

- *remove()* – цей метод видаляє перший знайдений елемент у списку зі значенням `*x*`.

```
my_list = [1, 2, 3, 2]
my_list.remove(2)
# my_list тепер буде [1, 3, 2]
```

- *pop()* – цей метод видаляє та повертає елемент за певним індексом (або останній елемент, якщо індекс не вказано).

```
my_list = [1, 2, 3]
popped_element = my_list.pop(1)  # Видаляє та повертає елемент з індексом 1 (2)
# my_list тепер буде [1, 3]
```

- *del* – можна використовувати для видалення елемента за певним індексом або навіть зрізу елементів.

```
my_list = [1, 2, 3]
del my_list[1]  # Видаляє елемент з індексом 1 (2)
# my_list тепер буде [1, 3]
```

Ці методи дають змогу змінювати списки в місці, що робить їх дуже потужним інструментом для роботи з даними в Python.

10.7. Пошук елемента списку й одержання відомостей про значення, які входять у список

У Python існують декілька методів для пошуку елементів у списку та одержання відомостей про їх значення.

1. *index()* – цей метод повертає індекс першого входження певного значення у списку. Якщо значення не знайдено, генерується помилка *ValueError*.

```
my_list = [1, 2, 3, 4, 5]
index = my_list.index(3)  # Повертає індекс 2
```

2. *count()* – цей метод повертає кількість входжень певного значення у списку.

```
my_list = [1, 2, 3, 2, 4, 2]
count = my_list.count(2)  # Повертає 3
```

3. Перевірка наявності елемента:

Можна використовувати оператор *in* для перевірки наявності певного значення у списку. Він поверне логічне значення *True*, якщо значення знайдено, та *False*, якщо ні.

```
my_list = [1, 2, 3, 4, 5]
is_present = 3 in my_list  # Повертає True
```

4. Отримання унікальних значень:

Якщо потрібно отримати лише унікальні значення у списку, можна використовувати конструктор множини для цього. Множина в Python не містить дублікатів.

```
my_list = [1, 2, 3, 2, 4, 2]
unique_values = list(set(my_list))  # Повертає [1, 2, 3, 4]
```

Ці методи дають змогу легко й ефективно працювати зі списками та отримувати відомості про їх значення.

10.8. Визначення кількості елементів

Для визначення кількості елементів у списку в Python можна використовувати вбудовану функцію *len()*. Вона повертає кількість елементів у списку.

```
# Приклад 1: Визначення кількості елементів у списку
my_list = [1, 2, 3, 4, 5]
length = len(my_list)  # length буде 5
string_list = ['apple', 'banana', 'cherry']
length = len(string_list)  # length буде 3
```

```
# Приклад 2: Визначення кількості елементів у порожньому списку
empty_list = []
length = len(empty_list)  # length буде 0

# Приклад 3: Визначення кількості елементів у вкладеному списку
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
length = len(nested_list)  # length буде 3
```

10.9. Перевертання та перемішування списку

У Python існують методи для перевертання та перемішування списків.

1. Перевертання списку:

Для перевертання списку використовується метод *reverse()*. Він змінює порядок елементів у списку на протилежний.

```
my_list = [1, 2, 3, 4, 5]
my_list.reverse()
# my_list тепер буде [5, 4, 3, 2, 1]
```

2. Перемішування списку:

Для перемішування елементів у списку використовується функція *shuffle()* з модуля *random*. Цей метод перемішує елементи у списку у випадковому порядку.

```
import random

my_list = [1, 2, 3, 4, 5]
random.shuffle(my_list)
# my_list тепер буде, наприклад, [3, 5, 1, 4, 2] (випадковий порядок)
```

10.10. Вибір елементів списку випадковим чином за допомогою функцій модуля random

Для вибору елементів списку випадковим чином за допомогою функцій модуля *random* у Python можна використовувати функцію *choice()* або *sample()*. Вони корисні, коли Вам потрібно проводити випадкові вибори або рандомізувати дані.

1. Функція *choice():*

Ця функція вибирає один елемент зі списку випадковим чином.

```
import random

my_list = [1, 2, 3, 4, 5]
random_element = random.choice(my_list)
```

2. Функція *sample():*

Ця функція вибирає кілька випадкових елементів зі списку. Вона приймає два аргументи: список та кількість елементів, які потрібно вибрати.

```
import random

my_list = [1, 2, 3, 4, 5]
random_elements = random.sample(my_list, k=3)  # Вибирає 3 випадкових елементи
```

10.11. Сортування списку

У Python для сортування списку використовуються методи та функції, які дають змогу впорядковувати елементи списку у зростаючому або спадаючому порядку.

1. Метод *sort():*

Цей метод сортує список на місці, змінюючи його порядок без повернення нового списку. Він сортує список у зростаючому

порядку за замовчуванням, але може бути використаний із параметром *reverse=True* для сортування у спадаючому порядку.

```
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5]
my_list.sort()  # Сортування у зростаючому порядку
```

2. Функція *sorted()*:

Ця функція приймає список як аргумент і повертає новий список, який містить елементи, відсортовані у зростаючому порядку. Вона не змінює оригінальний список.

```
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5]
sorted_list = sorted(my_list)
# Створення нового списку з відсортованими елементами
```

3. Сортування з використанням ключа:

Іноді ви хочете відсортувати список за допомогою ключа, наприклад, за довжиною рядків або якоюсь іншою характеристикою. Для цього ви можете використовувати параметр ***key***.

```
my_list = ['banana', 'apple', 'cherry']
sorted_list = sorted(my_list, key=len)  # Сортування за довжиною рядків
```

10.12. Заповнення списку числами

У Python є кілька способів заповнення списку числами.

1. Заповнення списку за допомогою циклу ***for***:

Ви можете використовувати цикл *for*, щоб заповнити список числами, які ви обчислюєте в цьому циклі.

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

2. Використання спискового виразу:

У Python є так звані спискові вирази, які дають змогу створювати списки швидко та лаконічно.

```
my_list = [i for i in range(1, 11)]
```

3. Заповнення списку за допомогою функції *range():*

Функція *range()* генерує послідовність чисел, які можна використовувати для заповнення списку.

```
my_list = list(range(1, 11))
```

4. Заповнення списку за допомогою іншого списку або генератора:

Ви також можете скопіювати вже існуючий список або використати генератор для створення списку з числами.

```
existing_list = [1, 2, 3, 4, 5]
new_list = existing_list[:]  # Копіювання списку
```

Або використайте генератор:

```
my_list = list(range(1, 11))  # Створення списку чисел від 1 до 10
```

10.13. Перетворення списку на рядок

У Python ви можете легко перетворити список на рядок за допомогою рядкового методу *join()*.

```
my_list = ['apple', 'banana', 'cherry']
my_string = ', '.join(my_list)
```

У цьому прикладі ми використали роздільник `,` для об’єднання кожного елемента списку в рядок. Результатом буде рядок, у якому кожен елемент списку розділений роздільником `,`. Ви можете використовувати будь-який інший роздільник замість `,` за вашим вибором.

Якщо ваш список містить числа, ви можете спочатку перетворити їх у рядки за допомогою функції map():

```
my_list = [1, 2, 3, 4, 5]
my_string = ', '.join(map(str, my_list))
```

Це перетворить кожне число у списку на рядок, а потім об'єднає їх за допомогою роздільника `,`.

10.14. Функція rang() та її параметри

У Python функція *range()* використовується для генерації послідовностей чисел. Вона має кілька параметрів, які визначають початок, кінець та крок послідовності. Детальний опис параметрів функції *range()*:

1. Початок (*start*): Параметр, який вказує початкове значення послідовності чисел. Якщо цей параметр не вказано, він автоматично приймає значення 0.

2. Кінець (*stop*): Параметр, який вказує кінцеве значення послідовності чисел. Послідовність не включає це значення.

3. Крок (*step*): Параметр, який вказує крок (інтервал) між числами у послідовності. За замовчуванням крок має значення 1.

Загальний синтаксис функції *range(): range(start, stop, step)*. Де *start, stop* і *step* – цілі числа.

Наприклад, якщо потрібно створити послідовність чисел від 0 до 9, можна скористатися функцією *range()* так:

```
my_range = range(0, 10)
```

Це створить послідовність чисел від 0 до 9 (не включно).

Також можна вказати крок для послідовності:

```
even_numbers = range(0, 11, 2)
```

Це створить послідовність чисел 0, 2, 4, 6, 8, 10.

10.15. Генератори списків та вирази-генератори

У Python генератор списку – це компактний спосіб створення списку через використання ітерації над якимось ітерабельним об'єктом, таким як інший список, кортеж або діапазон. Вираз генератора подібний до генератора списку, але повертає генераторний об'єкт, що може бути використаний для лінивого обчислення значень по одному, а не створення всього списку одразу. Ось приклади:

1. Генератор списку:

```
my_list = [x for x in range(10)]
# Або можна написати так:
# my_list = list(x for x in range(10))
```

У цьому прикладі створено список, що містить числа від 0 до 9.

2. Вираз генератора:

```
my_generator = (x for x in range(10))
```

У цьому прикладі створено генераторний об'єкт, який можна використовувати для отримання чисел від 0 до 9 «лінивим» способом, тобто по одному значенню за раз.

Головна відмінність між генератором списку та виразом генератора полягає в тому, що генератор списку миттєво створює весь список, тоді як вираз генератора створює генераторний об'єкт, який може бути використаний для генерації значень «лінивим» способом. Це означає, що значення генеруються лише в разі звертання до них, що може бути корисно для обробки великих обсягів даних без необхідності зберігання всього списку в пам'яті.

10.16. Властивості та параметри методу index()

Метод *index()* у Python використовується для знаходження першого входження певного значення в списку та повернення його індексу. Властивості та параметри цього методу:

1. Значення (*value*): Обов'язковий параметр, який вказує на значення, індекс якого потрібно знайти у списку.

2. Початковий індекс (*start*): Необов'язковий параметр, який вказує початковий індекс для пошуку. Пошук починається з цього індексу. Якщо не вказано, пошук починається з початку списку.

3. Кінцевий індекс (*end*): Необов'язковий параметр, який вказує кінцевий індекс для пошуку. Пошук ведеться до цього індексу, але не включає його. Якщо не вказано, пошук ведеться до кінця списку.

Якщо значення не знайдено у списку, метод *index()* викине виключення *ValueError*.

```
my_list = [10, 20, 30, 40, 50]
index = my_list.index(30)  # Повертає 2, оскільки 30 знаходиться на індексі 2
```

Якщо потрібно здійснити пошук у конкретному діапазоні індексів, можна використовувати параметри *start* та *end*.

```
my_list = [10, 20, 30, 40, 50]
index = my_list.index(30, 1, 4)
# Пошук 30 у діапазоні від індексу 1 до 4 (не включаючи 4)
```

У цьому випадку результат також буде 2, оскільки 30 знаходиться на індексі 2, але пошук ведеться в межах діапазону від 1 до 4.

10.17. Визначення параметрів елементів списку

У Python параметри елементів списку зазвичай описують їх структуру, властивості та характеристики. Оскільки у списку можуть бути елементи різних типів даних, параметри можуть відрізнятися.

1. Значення – це основна інформація, яка зберігається у списку.

2. Тип даних – визначає, які типи об'єктів можуть бути збережені у списку. Наприклад, список може містити цілі числа, рядки, списки, кортежі тощо.

3. Розмір – це кількість елементів, які містяться у списку.

4. Індекс – індекс елемента у списку, що вказує на його позицію.

5. Доступ до елементів – можливість доступу до окремих елементів у списку за їхніми індексами.

6. Методи та функції – списки мають різноманітні методи та функції, які можна використовувати для роботи з ними, такі як додавання, видалення, сортування тощо.

7. Змінність (*mutability*) – списки в Python є змінюваними об'єктами, тобто їх можна змінювати після створення. Це означає, що ви можете змінювати, додавати або видаляти елементи.

8. Ітерабельність (*iterability*) – списки є ітерабельними, тобто ви можете перебирати їх елементи у циклі або використовувати ітератори для доступу до кожного елемента.

Ці параметри дають змогу краще розуміти й працювати з елементами списку в Python.

Перелік питань для самоконтролю

1. Властивості списків у мові Python.
2. Визначення списку.
3. Способи створення списку.
4. Способи створення копії списку.
5. Операції над списками.
6. Перебір елементів списку.

7. Методи додавання та видалення елементів списку.

8. Пошук елемента списку й одержання відомостей про значення, які входять у список.

9. Визначення кількості елементів.

10. Сортування списку.

Тестові питання

1. Що таке список у мові Python:

а) послідовність елементів, яку можна змінювати;

б) колекція елементів, яку неможливо змінювати;

в) статична структура даних;

г) кортеж елементів.

2. Яка правильна форма створення порожнього списку в Python:

а) list();

б) [ ];

в) { };

г) tuple().

3. Як можна звертатися до елементів списку за їх індексом у Python:

а) за допомогою оператора `@`;

б) list[index];

в) list(index);

г) list.get(index).

4. Яка функція використовується для додавання нового елемента в кінець списку в Python:

а) add();

б) append();

в) insert();

г) extend().

5. Як можна перевірити наявність елемента у списку в Python:

а) contains();

б) in;

в) check();

г) exists().

6. Яка операція видаляє елемент із списку за його значенням у Python:

а) remove();

б) pop();

в) delete();

г) discard().

7. Яка операція видаляє елемент зі списку за його індексом у Python:

а) remove();

б) pop();

в) delete();

г) discard().

8. Яка функція повертає кількість елементів у списку в Python:

а) size();

б) count();

в) length();

г) len().

9. Як можна відсортувати список у Python у зростаючому порядку:

а) sort();

б) ascending();

в) sorted();

г) order().

10. Яка операція об’єднує два списки в Python:

а) join();

б) concatenate();

в) merge();

г) extend().

Рекомендовані джерела:

Основні: 1; 2; 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть програму, яка створює список з п’яти елементів, вводячи їх з клавіатури, а потім виводить цей список на екран.

2. Створіть програму, яка створює список з трьох чисел, а потім виводить другий елемент списку.

3. Напишіть програму, яка створює два списки, зливає їх в один, а потім виводить об’єднаний список.

4. Створіть програму, яка створює список з чотирьох рядків, змінює третій рядок на новий, введений з клавіатури, і виводить оновлений список.

5. Напишіть програму, яка створює список з п’яти чисел, видаляє третій елемент і виводить оновлений список.

6. Створіть програму, яка створює список з п’яти елементів і перевіряє, чи є заданий елемент у цьому списку.

7. Напишіть програму, яка створює список з п’яти чисел, сортує його в порядку зростання та виводить відсортований список.

8. Створіть програму, яка створює список з п’яти рядків, а потім виводить цей список у зворотному порядку.

9. Напишіть програму, яка створює список з чисел від 1 до 10 і виводить кількість елементів у цьому списку.

10. Створіть програму, яка запитує в користувача рядок слів, розділених пробілами, і розбиває цей рядок на список слів.

11. Напишіть програму, яка створює 3x3 матрицю (список списків), заповнену нулями, і виводить її на екран.

12. Створіть програму, яка запитує в користувача значення для кожного елемента 2x2 матриці та виводить її на екран.

13. Напишіть програму, яка створює 2x3 матрицю і виводить елемент у позиції (1, 2).

14. Створіть програму, яка транспонує 3x2 матрицю (перетворює рядки на стовпці) і виводить результат.

15. Напишіть програму, яка обчислює суму всіх елементів 2x3 матриці.

16. Створіть програму, яка знаходить максимальний елемент у 4x4 матриці.

17. Напишіть програму, яка створює 3x3 матрицю і виводить її у вигляді таблиці.

18. Створіть програму, яка перевіряє, чи є 3x3 матриця симетричною (матриця дорівнює своїй транспонованій).

19. Напишіть програму, яка сортує кожен рядок 2x3 матриці у зростаючому порядку.

20. Створіть програму, яка обчислює суму кожного стовпця 3x3 матриці.

21. Напишіть програму, яка створює список з трьох чисел, додає новий елемент у кінець списку і виводить оновлений список.

22. Створіть програму, яка створює список з п’яти рядків і вставляє новий рядок у середину списку.

23. Напишіть програму, яка створює список з п’яти чисел, видаляє перше входження заданого значення і виводить оновлений список.

24. Створіть програму, яка створює список з семи елементів і видаляє елемент за заданим індексом.

25. Напишіть програму, яка створює список з чисел і видаляє всі входження заданого числа.

Тема 11
Кортежі та множини

11.1. Означення, властивості та застосування кортежів множин та діапазонів.

11.2. Незмінювані послідовності типу кортеж.

11.3. Основна властивість типу «множина».

11.4. Особливості змінюваних і незмінюваних множин.

11.5. Методи створення кортежів. Одержання елемента кортежу за індексом.

11.6. Створення множини за допомогою функції set().

11.7. Перебір елементів множини в циклі for.

11.8. Методи для роботи з множинами.

11.9. Незмінювані множини типу frozentset.

11.10. Оператори та методи, що підтримують перетворення даних типу frozenset.

11.11. Параметри та формат функції range для задавання діапазону.

11.12. Функції, що підтримують перетворення даних типу «range».

11.13. Основні властивості діапазонів.

11.14. Означення та застосування операторів на множинах, які виконують логічні перетворення множин.

11.15. Генератори множин та їх синтаксис.

11.16. Ітератори, що представлені функціями модуля itertools.

11.1. Означення, властивості та застосування кортежів множин та діапазонів

Кортежі, множини та діапазони є вбудованими типами даних у Python і мають свої властивості та застосування.

1. Кортежі (*tuples*):

Кортеж – це незмінний тип даних, що містить послідовність елементів. Елементи кортежу можуть бути різних типів даних, і доступ до них здійснюється за допомогою індексів.

Основні властивості. Незмінність, ітерабельність, можливість вкладення.

Застосування. Кортежі використовуються для зберігання незмінюваних послідовностей даних, які не потребують зміни у процесі виконання програми. Вони часто використовуються для повернення декількох значень з функцій, передачі аргументів у функції та створення структурних даних.

2. Множини (*sets*):

Множина – це колекція унікальних та неупорядкованих елементів. Вона не має дублікатів і не підтримує індексації.

Основні властивості. Унікальність, неупорядкованість, ітерабельність.

Застосування. Множини використовуються для видалення дублікатів з колекцій, перевірки належності елементів до множини, операцій об'єднання, перетину та різниці між множинами.

3. Діапазони (*ranges*):

Діапазон – це послідовність чисел, яка представлена у вигляді діапазону. Він не зберігає всі свої значення в пам'яті, але генерує їх потрібною кількістю в разі звернення.

Основні властивості. «Лінива» оцінка (не вираховуються всі значення одразу), ітерабельність.

Застосування. Діапазони використовуються для представлення послідовностей чисел у деякому діапазоні. Вони корисні для ітерації в циклах, обрізання списків за допомогою зрізів та інших подібних випадків, де потрібно представити послідовність чисел у заданому діапазоні.

Ці типи даних дають змогу працювати з різними видами даних у Python зручним та ефективним способом, відповідно до потреб програми.

11.2. Незмінювані послідовності типу кортеж

Незмінні послідовності, такі як кортежі в Python, є колекціями даних, які не можна змінювати після їх створення.

1. Створення кортежу:

```
my_tuple = (1, 2, 3, 4, 5)
```

2. Доступ до елементів кортежу за допомогою індексів:

```
print(my_tuple[0])  # Виводить: 1
```

3. Розпакування кортежу:

```
a, b, c = my_tuple[:3]
print(a, b, c)  # Виводить: 1 2 3
```

4. Перевірка належності до кортежу:

```
if 4 in my_tuple:
    print("4 є у кортежі")
```

5. Повернення декількох значень з функції:

```
def get_coordinates():
    return (10, 20)

x, y = get_coordinates()
print(x, y)  # Виводить: 10 20
```

6. Створення кортежу з одного елемента:

```
single_element_tuple = (42,)
```

7. Створення кортежу з функції *zip():*

```
tuples = [(1, 'a'), (2, 'b'), (3, 'c')]
list1, list2 = zip(*tuples)
print(list1)
print(list2)
```

8. Створення кортежу з розширеного синтаксису:

```
    my_tuple = 1, 2, 3
```

11.3. Основна властивість типу «множина»

Основна властивість типу «множина» в Python полягає в тому, що множина є колекцією унікальних елементів, які не повторюються. Це означає, що в множині не може бути дублікатів значень. Коли ви додаєте елемент до множини, вона автоматично перевіряється на наявність цього елемента, і якщо він вже є в множині, то додавання не відбувається.

Основні властивості множин у Python:

1. Унікальність. Множина не може містити дублікатів. Якщо ви додаєте значення до множини, і це значення вже присутнє, то нічого не відбудеться.

2. Неупорядкованість. Елементи множини не мають фіксованого порядку. У разі виведення множини порядок елементів може змінюватися.

3. Швидкий доступ до елементів. Множина підтримує швидкий доступ до своїх елементів. Ви можете швидко перевірити належність елемента до множини.

4. Можливість використання для виконання операцій алгебри множин. Множини в Python підтримують операції об’єднання, перетину, різниці та симетричної різниці, що дає змогу легко виконувати різноманітні операції над ними.

Множини корисні для різних завдань, таких як видалення дублікатів зі списку, перевірка належності елемента до колекції, виконання операцій над множинами тощо. Вони дають змогу ефективно вирішувати такі завдання завдяки своїм основним властивостям.

11.4. Особливості змінюваних і незмінюваних множин

У Python існують два типи множин: змінювані (*mutable*) та незмінювані (*immutable*).

1. Змінювані множини (*mutable sets*):

Змінювані множини можна змінювати після їх створення. Змінювані множини представлені типом *set*. Змінювані множини підтримують операції додавання (*add()*), видалення (*remove()*), оновлення (*update()*) тощо.

2. Незмінювані множини (*immutable sets*):

Незмінювані множини не можна змінювати після їх створення. Це означає, що їх вміст не може бути змінений, але вони підтримують багато операцій, які не змінюють їх. Незмінювані множини представлені типом *frozenset.* Незмінювані множини підтримують операції порівняння, перетину, об'єднання, різниці тощо, але не підтримують операцій, які змінюють їх, наприклад, додавання або видалення елементів.

```
# Змінювані множини
mutable_set = set([1, 2, 3])
mutable_set.add(4)
mutable_set.remove(2)
print(mutable_set)  # Виводить: {1, 3, 4}

# Незмінювані множини
immutable_set = frozenset([1, 2, 3])
# Наступний рядок викличе помилку:
# immutable_set.add(4)
# Виведе помилку TypeError: 'frozenset' object does not support item assignment
```

Обидва типи множин корисні в різних сценаріях, залежно від того, чи потрібно змінювати вміст множини після її створення.

11.5. Методи створення кортежів. Одержання елемента кортежу за індексом

У Python існує кілька способів створення кортежів і отримання елементів за їх індексом.

1. Створення порожнього кортежу:

```
empty_tuple = ()
```

2. Створення кортежу з використанням літералу:

```
my_tuple = (1, 2, 3, 4, 5)
```

3. Створення кортежу без використання дужок:

```
my_tuple = 1, 2, 3, 4, 5
```

4. Створення кортежу з одного елемента:

```
single_element_tuple = (42,)
```

5. Створення кортежу з ітерабельного об’єкта:

```
my_list = [1, 2, 3]
my_tuple = tuple(my_list)
```

Отримання елементів кортежу за їх індексом:

```
my_tuple = (10, 20, 30, 40, 50)
print(my_tuple[0])  # Отримання першого елемента. Виведе: 10
print(my_tuple[2])  # Отримання третього елемента. Виведе: 30
```

У Python індекси елементів кортежу починаються з нуля, тому *my_tuple[0]* поверне перший елемент кортежу, а *my_tuple[2]* поверне третій елемент.

11.6. Створення множини за допомогою функції set()

У Python множини можна створити за допомогою функції *set()*. Ця функція приймає ітерабельний об’єкт (такий як список, кортеж або рядок) і створює нову множину, яка містить унікальні елементи цього об’єкта.

```
# Створення множини зі списку
my_list = [1, 2, 3, 4, 4, 5, 5]
my_set = set(my_list)
print(my_set)  # Виведе: {1, 2, 3, 4, 5}

# Створення множини з кортежу
my_tuple = (1, 2, 3, 4, 4, 5, 5)
my_set = set(my_tuple)
print(my_set)  # Виведе: {1, 2, 3, 4, 5}

# Створення множини з рядка
my_string = "hello"
my_set = set(my_string)
print(my_set)  # Виведе: {'h', 'e', 'l', 'o'}
```

Зауважимо, що в усіх цих прикладах дублікати були вилучені, і в результаті ми отримали множину, яка містить тільки унікальні елементи.

11.7. Перебір елементів множини в циклі for

У Python множини є ітерабельними об’єктами, тому ви можете перебирати їх елементи за допомогою циклу *for*.

```
my_set = {1, 2, 3, 4, 5}

# Перебір елементів множини
for element in my_set:
    print(element)
```

Цей код перебирає кожен елемент у множині `my_set` і виводить його на екран. Вивід буде таким:

```
1
2
3
4
5
```

Цей метод є зручним для виконання операцій над кожним елементом множини або для виконання якихось дій для кожного елемента. Наприклад, можна виконати певні дії з кожним елементом множини або використати їх для порівнянь з іншими даними.

11.8. Методи для роботи з множинами

У Python існують різні методи для роботи з множинами, які дають змогу виконувати різноманітні операції над ними.

1. Метод *add(element)* – додає новий елемент до множини.

2. Метод *remove(element)* – видаляє вказаний елемент із множини. Якщо елемент відсутній, видає помилку.

3. Метод *discard(element)* – видаляє вказаний елемент із множини. Якщо елемент відсутній, жодної помилки не видає.

4. Метод *in* – перевіряє, чи належить елемент множині.

5. Метод *clear()* – видаляє всі елементи з множини.

6. Метод *union(other_set),* `|` – повертає нову множину, що містить усі елементи обох множин.

7. Метод *intersection(other_set)*, `&` – повертає нову множину, яка містить елементи, що присутні в обох множинах.

8. Метод *difference(other_set)*, `-` – повертає нову множину, яка містить елементи з першої множини, що не присутні у другій.

9. Симетрична різниця множин. *symmetric_difference(other_set)*, `^` – повертає нову множину, яка містить елементи, що присутні в одній множині, але відсутні в іншій.

10. Метод *copy()* – створює копію множини.

11.9. Незмінювані множини типу frozentset

У Python *frozenset* є типом, який представляє незмінні множини. Це означає, що елементи *frozenset* не можна змінювати після їх створення. Вони аналогічні звичайним множинам, але мають обмежену функціональність через свою незмінюваність.

```
# Створення frozenset зі списку
my_list = [1, 2, 3, 4, 4, 5, 5]
my_frozenset = frozenset(my_list)
print(my_frozenset)  # Виведе: frozenset({1, 2, 3, 4, 5})

# Перевірка належності
print(2 in my_frozenset)  # Виведе: True

# Отримання довжини
print(len(my_frozenset))  # Виведе: 5

# Перебір елементів
for element in my_frozenset:
    print(element)
```

11.10. Оператори та методи, що підтримують перетворення даних типу frozenset

frozenset у Python є незмінним типом даних, тому він не має методів, які дають змогу його змінювати. Однак ви можете використовувати різні операції та функції для перетворення даних типу *frozenset*.

1. Перетворення списку в *frozenset*:

```
my_list = [1, 2, 3, 4, 5]
my_frozenset = frozenset(my_list)
```

2. Перетворення кортежу в *frozenset*:

```
my_tuple = (1, 2, 3, 4, 5)
my_frozenset = frozenset(my_tuple)
```

3. Копіювання існуючого *frozenset*:

```
original_frozenset = frozenset([1, 2, 3, 4, 5])
copied_frozenset = original_frozenset.copy()
```

4. Об’єднання двох *frozenset*:

```
frozenset1 = frozenset([1, 2, 3])
frozenset2 = frozenset([4, 5, 6])
combined_frozenset = frozenset1.union(frozenset2)
```

5. Перетин двох *frozenset*:

```
frozenset1 = frozenset([1, 2, 3])
frozenset2 = frozenset([2, 3, 4])
intersection_frozenset = frozenset1.intersection(frozenset2)
```

6. Різниця двох *frozenset*:

```
frozenset1 = frozenset([1, 2, 3])
frozenset2 = frozenset([3, 4, 5])
difference_frozenset = frozenset1.difference(frozenset2)
```

7. Симетрична різниця двох *frozenset*:

```
frozenset1 = frozenset([1, 2, 3])
frozenset2 = frozenset([3, 4, 5])
symmetric_difference_frozenset =
frozenset1.symmetric_difference(frozenset2)
```

Зверніть увагу, що всі ці операції та функції повертають новий об’єкт типу *frozenset*, оскільки *frozenset* є незмінним типом даних.

11.11. Параметри та формат функції range для задавання діапазону

Функція *range()* в Python створює послідовність чисел у заданому діапазоні. Вона має кілька форматів та параметрів, які дають змогу задавати діапазон за різними критеріями. Ось огляд параметрів і форматів функції *range()*:

1. Один аргумент. *range(stop)* – створює послідовність чисел від 0 до *stop - 1*.

2. Два аргументи. *range(start, stop)* – створює послідовність чисел від *start* до *stop - 1*.

3. Три аргументи. *range(start, stop, step)* – створює послідовність чисел від *start* до *stop - 1* з кроком *step*. Якщо крок не вказано, за замовчуванням він дорівнює 1.

```
# Послідовність чисел від 0 до 9
print(list(range(10)))  # Виведе: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Послідовність чисел від 2 до 9
print(list(range(2, 10)))  # Виведе: [2, 3, 4, 5, 6, 7, 8, 9]

# Послідовність парних чисел від 2 до 10
print(list(range(2, 11, 2)))  # Виведе: [2, 4, 6, 8, 10]
```

Ця функція корисна для ітерації через певний діапазон чисел у циклі або для створення списків чисел.

11.12. Функції, що підтримують перетворення даних типу «range»

Функція *range()* у Python створює об’єкт діапазону, а не список конкретних значень. Проте ви можете перетворити цей об’єкт у список або кортеж за допомогою вбудованих функцій *list()* або *tuple()*.

```
# Створення діапазону від 0 до 9 та перетворення його у список
my_range = range(10)
my_list = list(my_range)
print(my_list)  # Виведе: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Створення діапазону від 1 до 10 з кроком 2 та перетворення його в кортеж
my_range = range(1, 11, 2)
my_tuple = tuple(my_range)
print(my_tuple)  # Виведе: (1, 3, 5, 7, 9)
```

11.13. Основні властивості діапазонів

Діапазон у Python представлений об'єктом *range*. Він являє собою послідовність чисел і використовується для ітерації через певний діапазон значень.

1. Не зберігає всі елементи в пам'яті. Діапазони не зберігають усі елементи в пам'яті, тому вони ефективно використовують пам'ять, навіть якщо діапазон великий.

2. Не змінюється. Після створення діапазон не можна змінити. Ви не можете додати, видалити або змінити його елементи.

3. Використовується для ітерації. Головне призначення діапазонів – це створення послідовності чисел для ітерації через них у циклі.

4. Легко перетворюється у список або кортеж. Ви можете легко перетворити діапазон у список або кортеж, використавши функції *list()* або *tuple()*.

5. Параметризований. Діапазон може бути створений з указанням початкового значення, кінцевого значення та кроку. Якщо початкове значення не вказане, за замовчуванням воно дорівнює 0. Якщо крок не вказаний, за замовчуванням він дорівнює 1.

6. Ліниве обчислення. Елементи діапазону обчислюються лише тоді, коли до них звертаються. Це дає змогу ефективно використовувати пам'ять.

7. Може використовуватися для створення списків, кортежів та інших структур даних. Діапазони часто використовуються для створення списків, кортежів та інших структур даних, коли потрібно створити послідовність чисел.

Враховуючи ці властивості, діапазони є потужним інструментом для роботи з послідовностями чисел у Python, особливо коли потрібно ефективно створити ітеровану послідовність чисел.

11.14. Означення та застосування операторів на множинах, які виконують логічні перетворення множин

Оператори на множинах у Python дають змогу виконувати різні логічні операції на множинах, такі як об'єднання, перетин, різниця та симетрична різниця.

1. Об'єднання (`|` або *union()*). Об'єднання множин включає всі унікальні елементи з обох множин.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1 | set2  # Або можна використовувати: result = set1.union(set2)
print(result)  # Виведе: {1, 2, 3, 4, 5}
```

2. Перетин (`&` або *intersection()*). Перетин множин включає тільки ті елементи, які присутні в обох множинах.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1 & set2  # Або можна використовувати: result = set1.intersection(set2)
print(result)  # Виведе: {3}
```

3. Різниця (`-` або *difference()*). Різниця множин включає усі елементи, які є в першій множині, але не є в другій.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1 - set2  # Або можна використовувати: result = set1.difference(set2)
print(result)  # Виведе: {1, 2}
```

4. Симетрична різниця (`^` або *symmetric_difference()*). Симетрична різниця множин включає елементи, які є лише в одній з множин, але не в обох одночасно.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1 ^ set2  # Або можна використовувати: result = set1.symmetric_difference(set2)
print(result)  # Виведе: {1, 2, 4, 5}
```

11.15. Генератори множин та їх синтаксис

У Python генератори множин дають змогу створювати множини за допомогою компактного та ефективного синтаксису, який називається генераторами виразів. Синтаксис генераторів множин подібний до генераторів списків, але використовує фігурні дужки замість квадратних. Загальна форма генератора множин:

{вираз for змінна in послідовність [if умова]}

Де:

- *`вираз`* – вираз, який виконується для кожного елемента.
- *`змінна`* – змінна, яка приймає значення з послідовності.
- *`послідовність`* – послідовність елементів, яка використовується для створення множини.
- *`[if умова]`* (опціонально) – умова, яка фільтрує елементи.

```
# Створення множини з квадратами чисел від 0 до 9
my_set = {x2 for x in range(10)}
print(my_set)  # Виведе: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}

# Створення множини з парними числами від 0 до 9
even_numbers = {x for x in range(10) if x % 2 == 0}
print(even_numbers)  # Виведе: {0, 2, 4, 6, 8}
```

У цих прикладах ми створюємо множини за допомогою генераторів множин. Перший приклад створює множину квадратів чисел від 0 до 9, а другий – множину парних чисел від 0 до 9, використовуючи умову `if` для фільтрації лише парних чисел. Генератори множин – це короткий, елегантний і зручний спосіб створення множин у Python.

11.16. Ітератори, що представлені функціями модуля itertools

Модуль *itertools* у Python містить набір інструментів для роботи з ітераторами та послідовностями даних. Він містить деякі корисні функції, які дають змогу ефективно створювати, комбінувати та маніпулювати ітераторами.

1. *count(start=0, step=1)* – генерує послідовність чисел, починаючи з *start*, з кроком *step*.

```
from itertools import count

for i in count(10, 2):
    print(i)
    if i > 20:
        break
```

2. *cycle(iterable)* – безкінечно повторює послідовність з *iterable*.

```
from itertools import cycle

colors = cycle(['red', 'green', 'blue'])
for i in range(5):
    print(next(colors))
```

3. *repeat(elem, n=inf)* – повертає *elem n* разів. Якщо *n* не вказано, повертається безкінечна послідовність.

```
from itertools import repeat

for i in repeat('hello', 3):
    print(i)
```

4. *chain(*iterables)* – об'єднує послідовності в одну послідовність.

```
from itertools import chain

numbers = chain([1, 2, 3], [4, 5, 6], [7, 8, 9])
print(list(numbers))  # Виведе: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

5. *zip_longest(*iterables, fillvalue=None)* – паралельно об'єднує послідовності, заповнюючи пропуски значенням *fillvalue*.

```
from itertools import zip_longest

names = ['Alice', 'Bob', 'Charlie']
ages = [30, 25]
zipped = zip_longest(names, ages, fillvalue='Unknown')
print(list(zipped))   # Виведе: [('Alice', 30), ('Bob', 25), ('Charlie',
'Unknown')]
```

Ці функції – лише кілька прикладів того, що пропонує модуль *itertools*. Вони дають змогу ефективно створювати та маніпулювати ітераторами, що робить їх корисними для широкого спектра завдань програмування.

Перелік питань для самоконтролю

1. Властивості кортежів множин та діапазонів.
2. Незмінювані послідовності.
3. Основна властивість «множин».
4. Особливості незмінюваних множин.
5. Методи створення кортежів.
6. Одержання елемента кортежу за індексом.
7. Створення множини за допомогою функції **set().**
8. Перебір елементів множини в циклі **for.**
9. Методи для роботи з множинами.
10. Незмінювані множини типу **frozentset.**
11. Оператори та методи, що підтримують перетворення даних типу **frozenset.**

Тестові питання

1. Яка різниця між кортежем та списком у Python:

а) кортежі можна змінювати, а списки – ні;

б) кортежі не можна змінювати, а списки – можна;

в) обидва можна змінювати;

г) немає різниці між ними.

2. Яка правильна форма створення порожнього кортежу в Python:

а) tuple();

б) ();

в) { };

г) list().

3. Яка операція використовується для додавання елементів у множину в Python:

а) add();

б) insert();

в) append();

г) extend().

4. Яка функція перевіряє наявність елемента у кортежі в Python:

а) contains();

б) in;

в) check();

г) exists().

5. Як можна звертатися до елементів кортежу за їх індексом у Python:

а) за допомогою оператора `@`;

б) tuple[index];

в) tuple(index);

г) tuple.get(index).

6. Яка операція видаляє елемент із множини в Python:

а) remove();

б) pop();

в) delete();

г) discard().

7. Як можна відсортувати кортеж у Python:

а) sorted();

б) sort();

в) order();

г) sort(tuple).

8. Яка функція повертає кількість елементів у кортежі в Python:

а) size();
б) count();
в) length();
г) len().

9. Яка операція об’єднує два кортежі в Python:

а) join();
б) concatenate();
в) merge();
г) + (плюс).

10. Яка операція видаляє всі елементи з множини в Python:

а) clear();
б) remove_all();
в) delete_all();
г) discard_all().

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть програму, яка створює кортеж з трьох чисел і виводить перший і останній елемент кортежу.

2. Напишіть програму, яка перетворює кортеж на список, змінює один з його елементів, а потім знову перетворює його на кортеж.

3. Створіть кортеж, що містить рядок, число та список, і виведіть кожен елемент окремо.

4. Напишіть програму, яка створює два кортежі й об’єднує їх в один. Виведіть результат.

5. Створіть кортеж з одного елемента і повторіть цей кортеж три рази, виведіть результат.

6. Напишіть програму, яка перевіряє, чи є певний елемент у кортежі.

7. Створіть кортеж з трьох значень і розпакуйте його в три змінні, виведіть ці змінні.

8. Напишіть програму, яка створює кортеж з п’яти елементів і виводить його довжину.

9. Створіть кортеж з чисел і знайдіть максимальне і мінімальне значення.

10. Напишіть програму, яка підраховує кількість входжень певного елемента в кортеж.

11. Створіть пусту множину, додайте кілька елементів та виведіть множину.

12. Напишіть програму, яка створює множину з п’яти чисел, видаляє певний елемент і виводить оновлену множину.

13. Напишіть програму, яка перевіряє, чи є певний елемент у множині.

14. Створіть дві множини, об’єднайте їх і знайдіть перетин.

15. Напишіть програму, яка знаходить різницю між двома множинами.

Тема 12
Словники та методи роботи з ними

12.1. Означення, властивості та застосування словників.

12.2. Способи створення словників.

12.3. Операції над словниками: доступ до елементів словника, перевірка існування ключа.

12.4. Методи для роботи зі словниками.

12.5. Перебір елементів словника за допомогою циклу for.

12.6. Сортування.

12.1. Означення, властивості та застосування словників

Словники в Python – це структури даних, які використовуються для зберігання колекції пар ключ-значення. Вони є одними з найбільш корисних і часто використовуваних структур даних у Python.

Словник у Python – це колекція пар ключ-значення, де кожен ключ унікальний і відображається на своє значення. У словниках важливою одиницею є ключ, який використовується для доступу до значення. Ключі можуть бути майже будь-якими незмінюваними об’єктами, такими як рядки, числа або кортежі, а значення можуть бути будь-якими об’єктами Python.

Властивості:

1. Унікальність ключів. Кожен ключ у словнику є унікальним, тобто він може бути представлений лише один раз.

2. Змінність. Словники є змінюваними структурами даних, тобто їх можна змінювати після створення, додавати нові пари ключ-значення, видаляти та змінювати існуючі.

3. Неупорядкованість. Елементи словника не мають певного порядку. Ключі можуть зберігатися в словнику в довільному порядку.

Застосування:

1. Зберігання даних. Словники використовуються для зберігання даних у вигляді пар ключ–значення, наприклад, словники

можуть використовуватися для зберігання інформації про студентів (ключ – ім’я студента, значення – його оцінки).

2. Швидкий доступ. Завдяки швидкому доступу до елементів за ключем, словники використовуються для ефективного пошуку та отримання значень.

3. Моделювання об’єктів. Словники дають змогу створювати складні структури даних для моделювання об’єктів з реального світу.

Словники – це потужний інструмент у Python, який дає змогу ефективно робити багато операцій з даними, такі як пошук, зберігання та маніпулювання.

У Python ключі словника є унікальними, тобто вони можуть бути представлені лише один раз. Це схоже на асоціативні масиви в багатьох мовах програмування, де кожен ключ також є унікальним. Як і словники в Python, асоціативні масиви в багатьох мовах програмування є змінюваними структурами даних, тобто їх можна змінювати після створення. Синтаксис роботи зі словниками в Python може відрізнятися від синтаксису асоціативних масивів у інших мовах програмування. Наприклад, у мові JavaScript асоціативні масиви можна створювати за допомогою літералів `{}`, а доступ до значень можна отримати за допомогою квадратних дужок `[]`.

12.2. Способи створення словників

У Python існує кілька способів створення словників.

1. Літеральний спосіб. Використовує фігурні дужки `{}` для створення словника. Кожен запис у словнику представляється у вигляді пари ключ-значення, розділених двокрапкою `:`.

```
# Створення словника з трьома парами ключ-значення
my_dict = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

2. Функція *dict()*. Функція *dict()* приймає послідовність пар ключ-значення (часто список кортежів) і створює словник з них.

```
# Створення словника з використанням функції dict()
pairs = [('key1', 'value1'), ('key2', 'value2'), ('key3', 'value3')]
my_dict = dict(pairs)
```

3. Генератор словників. Генератор словників – це вираз, який створює словник за допомогою ітерації через послідовність елементів та виконання певної операції для кожного елементу.

```
# Створення словника за допомогою генератора словників
my_dict = {key: value for key, value in [('key1', 'value1'), ('key2', 'value2'), ('key3', 'value3')]}
```

4. Використання методу *fromkeys()*. Метод *fromkeys()* створює новий словник з заданими ключами та однаковим значенням для кожного ключа.

```
# Створення словника з використанням методу fromkeys()
keys = ['key1', 'key2', 'key3']
value = 'default_value'
my_dict = dict.fromkeys(keys, value)
```

12.3. Операції над словниками: доступ до елементів словника, перевірка існування ключа

Операції над словниками в Python дають змогу отримувати доступ до елементів словника та перевіряти існування ключів.

1. Доступ до елементів словника за ключем. Для отримання значення за певним ключем використовується оператор `[]`. Він дає змогу звертатися до значення, яке відповідає вказаному ключу. Якщо ключ відсутній у словнику, виникає помилка *KeyError*.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Доступ до значення за ключем 'a'
value_a = my_dict['a']  # value_a = 1

# Доступ до значення за ключем 'b'
value_b = my_dict.get('b')  # value_b = 2
```

2. Перевірка існування ключа. Для перевірки того, чи існує певний ключ у словнику, можна використовувати оператор *in*. Він повертає *True*, якщо ключ присутній у словнику, і *False* – якщо відсутній.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Перевірка існування ключа 'a'
if 'a' in my_dict:
    print("Ключ 'a' існує у словнику.")
else:
    print("Ключ 'a' відсутній у словнику.")
```

Альтернативно можна використовувати метод *get()*, який поверне значення ключа, якщо він існує, або *None*, якщо відсутній.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Перевірка існування ключа 'd' за допомогою методу get()
value_d = my_dict.get('d')
if value_d is not None:
    print("Ключ 'd' існує у словнику.")
else:
    print("Ключ 'd' відсутній у словнику.")
```

Ці операції дають змогу зручно працювати з елементами словника та перевіряти наявність ключів перед доступом до них, що допомагає уникнути помилок у виконанні програм.

Вилучення елементів словника**.** Операція вилучення елементів зі словника виконується за допомогою ключового слова *del*. Це дає змогу вилучати конкретні елементи за їх ключем.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Вилучення елемента з ключем 'b'
del my_dict['b']

print(my_dict)  # {'a': 1, 'c': 3}
```

У цьому прикладі елемент з ключем *'b'* був вилучений зі словника *my_dict*. Після виконання операції *del* елемент більше не буде доступний у словнику.

Також можна використовувати метод *pop()*, який вилучає та повертає значення видаленого ключа. Якщо ключ не знайдено, він може повернути задане значення за замовчуванням.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Вилучення елемента з ключем 'b' за допомогою методу pop()
value_b = my_dict.pop('b')

print(my_dict)   # {'a': 1, 'c': 3}
print(value_b)   # 2
```

12.4. Методи для роботи зі словниками

У Python існує кілька методів, які допомагають працювати зі словниками.

1. Метод clear() – використовується для видалення всіх елементів зі словника.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
my_dict.clear()
print(my_dict)  # {}
```

2. Метод *copy()* – створює копію словника.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
new_dict = my_dict.copy()
print(new_dict)  # {'a': 1, 'b': 2, 'c': 3}
```

3. Метод *keys()* – повертає перелік усіх ключів у словнику.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
keys = my_dict.keys()
print(keys)  # dict_keys(['a', 'b', 'c'])
```

4. Метод *values()* – повертає перелік усіх значень у словнику.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
values = my_dict.values()
print(values)  # dict_values([1, 2, 3])
```

5. Метод *items()* – повертає перелік кортежів, що містять пари ключ-значення у словнику.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
items = my_dict.items()
print(items)  # dict_items([('a', 1), ('b', 2), ('c', 3)])
```

6. Метод *update()* – додає пари ключ-значення з одного словника в інший.

```
my_dict = {'a': 1, 'b': 2}
new_dict = {'c': 3, 'd': 4}
my_dict.update(new_dict)
print(my_dict)  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

7. Метод *pop()* – вилучає та повертає значення для вказаного ключа. Якщо ключ відсутній, можна вказати значення за замовчуванням, яке поверне метод.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
value = my_dict.pop('b')
print(value)  # 2
print(my_dict)  # {'a': 1, 'c': 3}
```

Генератори словників. Генератор словників – це структура даних у Python, яка дає змогу створювати словники за допомогою зручного синтаксису. Генератор словників дає змогу створювати словники за допомогою виразу, який зазвичай містить ключі та відповідні їм значення. Основна синтаксична форма генератора словників виглядає так:

{ключ: вираз for змінна in послідовність}

де *`ключ`* – ключ, що буде доданий до словника, *`вираз`* – вираз, що визначає значення ключа, *`змінна`* – змінна, що ітерується по послідовності, і *`послідовність`* – послідовність значень, яка використовується для створення ключів.

1. Створення словника зі списку:

```
my_list = ['a', 'b', 'c']
my_dict = {item: len(item) for item in my_list}
print(my_dict)  # {'a': 1, 'b': 1, 'c': 1}
```

2. Створення словника на основі умовного виразу:

```
numbers = [1, 2, 3, 4, 5]
even_odd_dict = {num: ('even' if num % 2 == 0 else 'odd') for num in numbers}
print(even_odd_dict)  # {1: 'odd', 2: 'even', 3: 'odd', 4: 'even', 5: 'odd'}
```

3. Створення словника на основі іншого словника:

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
doubled_dict = {key: value * 2 for key, value in my_dict.items()}
print(doubled_dict)  # {'a': 2, 'b': 4, 'c': 6}
```

Визначення кількості ключів. Для визначення кількості ключів у словнику використовується вбудована функція *len()*. Вона повертає кількість елементів у словнику, тобто кількість його ключів.

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Визначення кількості ключів у словнику
num_keys = len(my_dict)

print("Кількість ключів у словнику:", num_keys)
# Кількість ключів у словнику: 3
```

Цей код визначає кількість ключів у словнику *my_dict* та друкує її значення. У результаті отримаємо кількість ключів у словнику, яка буде виведена на екран.

12.5. Перебір елементів словника за допомогою циклу for

Перебір елементів словника за допомогою циклу *for* дає змогу пройтися по кожній парі ключ-значення у словнику. Для цього можна використовувати методи *keys(), values()* або *items()*.

1. Перебір ключів:

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

for key in my_dict.keys():
    print(key)
```

2. Перебір значень:

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

for value in my_dict.values():
    print(value)
```

3. Перебір пар ключ-значення:

```
my_dict = {'a': 1, 'b': 2, 'c': 3}

for key, value in my_dict.items():
    print(key, value)
```

12.6. Сортування

У Python для сортування словника можна використовувати метод *sorted()* у поєднанні з методом *items()*.

1. Сортування словника за ключами.

```
# Оригінальний словник
original_dict = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

# Сортування словника за ключами
sorted_dict = dict(sorted(original_dict.items()))

print(sorted_dict)
```

Результат:

```
{'apple': 4, 'banana': 3, 'orange': 2, 'pear': 1}
```

У цьому прикладі:

- *original_dict.items()* повертає список пар (ключ, значення) зі словника.
- *sorted()* сортує ці пари за ключами (оскільки це перший елемент кожної пари).
- *dict()* перетворює відсортовані пари назад у словник.

2. Сортування словника за ключами у зворотному порядку.

```
# Оригінальний словник
original_dict = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

# Сортування словника за ключами у зворотному порядку
sorted_dict = dict(sorted(original_dict.items(), reverse=True))

print(sorted_dict)
```

Результат:

```
{'pear': 1, 'orange': 2, 'banana': 3, 'apple': 4}
```

У цьому разі параметр *reverse=True* змінює порядок на зворотний.

3. Сортування словника за значеннями.

Іноді може знадобитися сортувати словник не за ключами, а за значеннями.

```
# Оригінальний словник
original_dict = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

# Сортування словника за значеннями
sorted_dict_by_value = dict(sorted(original_dict.items(),
key=lambda item: item[1]))

print(sorted_dict_by_value)

# Результат: {'pear': 1, 'orange': 2, 'banana': 3, 'apple': 4}
```

У цьому разі:

- *key=lambda item: item[1]* вказує *sorted()* використовувати значення (другий елемент пари) для сортування.

4. Сортування словника за значеннями у зворотному порядку.

```
# Оригінальний словник
original_dict = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

# Сортування словника за значеннями у зворотному порядку
sorted_dict_by_value_desc = dict(sorted(original_dict.items(),
key=lambda item: item[1], reverse=True))

print(sorted_dict_by_value_desc)
# Результат: {'apple': 4, 'banana': 3, 'orange': 2, 'pear': 1}
```

Тут *reverse=True* застосовується до значень, що робить сортування за значеннями у зворотному порядку.

Функція *sort()* у Python використовується для сортування списків на місці, тобто безпосередньо змінює сам список і не повертає новий. Однак *sort()* не працює безпосередньо зі словниками, оскільки словники в Python не є списками. Проте можна перетворити словник у список пар (ключ-значення), відсортувати цей список, а потім створити новий словник із відсортованих пар.

1. Сортування словника за ключами за допомогою *sort()*:

- перетворення словника в список пар (ключ-значення);
- сортування списку за допомогою `sort()`;
- створення нового словника з відсортованих пар.

```
# Оригінальний словник
original_dict = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

# Перетворення словника в список пар (ключ-значення)
items_list = list(original_dict.items())

# Сортування списку за ключами
items_list.sort(key=lambda item: item[0])

# Створення нового словника з відсортованих пар
sorted_dict = dict(items_list)

print(sorted_dict)

# Результат: {'apple': 4, 'banana': 3, 'orange': 2, 'pear': 1}
```

2. Сортування словника за значеннями за допомогою *sort()*.

Якщо ви хочете відсортувати словник за значеннями, аналогічно можете перетворити словник у список пар, відсортувати його за значеннями, а потім створити новий словник.

```
# Оригінальний словник
original_dict = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

# Перетворення словника в список пар (ключ-значення)
items_list = list(original_dict.items())

# Сортування списку за значеннями
items_list.sort(key=lambda item: item[1])

# Створення нового словника з відсортованих пар
sorted_dict_by_value = dict(items_list)

print(sorted_dict_by_value)
# Результат: {'pear': 1, 'orange': 2, 'banana': 3, 'apple': 4}
```

3. Сортування словника за ключами у зворотному порядку за допомогою *sort()*.

```
# Оригінальний словник
original_dict = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

# Перетворення словника в список пар (ключ-значення)
items_list = list(original_dict.items())

# Сортування списку за ключами у зворотному порядку
items_list.sort(key=lambda item: item[0], reverse=True)

# Створення нового словника з відсортованих пар
sorted_dict_desc = dict(items_list)

print(sorted_dict_desc)
# Результат: {'pear': 1, 'orange': 2, 'banana': 3, 'apple': 4}
```

4. Сортування словника за значеннями у зворотному порядку за допомогою *sort()*.

```
# Оригінальний словник
original_dict = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

# Перетворення словника в список пар (ключ-значення)
items_list = list(original_dict.items())

# Сортування списку за значеннями у зворотному порядку
items_list.sort(key=lambda item: item[1], reverse=True)

# Створення нового словника з відсортованих пар
sorted_dict_by_value_desc = dict(items_list)

print(sorted_dict_by_value_desc)

# Результат: {'apple': 4, 'banana': 3, 'orange': 2, 'pear': 1}
```

Перелік питань для самоконтролю

1. Означення, властивості та застосування словників.
2. Способи створення словників.
3. Операції над словниками.
4. Вилучення елементів словника за функцією **del.**
5. Методи для роботи зі словниками.
6. Генератори словників.
7. Визначення кількості ключів.
8. Перебір елементів словника за допомогою циклу **for**.
9. Сортування за ключами.

Тестові питання

1. Як можна створити порожній словник у Python:

а) `{}`;

б) `dict()`;

в) `empty_dict()`;

г) `{}()`.

2. Яка операція дає змогу отримати значення за ключем у словнику:

а) `get()`;

б) `value()`;

в) `fetch()`;

г) `retrieve()`.

3. Яка функція видаляє пару ключ-значення зі словника за ключем:

а) `remove()`;

б) `popitem()`;

в) `delete()`;

г) `pop()`.

4. Яка операція додає нову пару ключ-значення у словник:

а) `add()`;

б) `insert()`;

в) `append()`;

г) `update()`.

5. Яка з наведених функцій перевіряє, чи існує певний ключ у словнику:

а) `exist()`;

б) `in()`;

в) `contains()`;

г) `haskey()`.

6. Яка функція повертає список усіх ключів у словнику:

а) `keys()`;

б) `get_keys()`;

в) `list_keys()`;

г) `all_keys()`.

7. Яка з наступних функцій повертає список усіх значень у словнику:

а) `get_values()`;

б) `values()`;

в) `list_values()`;

г) `all_values()`.

8. Яке ключове слово використовується для ітерації через пари ключ-значення у словнику:

а) `for`;

б) `iterate`;

в) `foreach`;

г) `loop`.

9. Яка функція повертає кількість пар ключ-значення у словнику:

а) `size()`;

б) `length()`;

в) `count()`;

г) `len()`.

10. Яка з наведених функцій перевіряє, чи є словник порожнім:

а) `empty()`;

б) `is_empty()`;

в) `isEmpty()`;

г) `none()`.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Створіть словник, який містить три пари «ключ-значення» (наприклад, ім'я-віку), і виведіть його на екран.
2. Напишіть програму, яка створює словник з кількома парами «ключ-значення» і виводить значення за заданим ключем.
3. Створіть словник, додайте до нього нову пару «ключ-значення» та виведіть оновлений словник.
4. Напишіть програму, яка створює словник з кількома парами «ключ-значення» та оновлює значення для заданого ключа.
5. Створіть словник, видаліть пару за заданим ключем та виведіть оновлений словник.
6. Напишіть програму, яка перевіряє, чи є певний ключ у словнику.
7. Створіть словник і використайте методи `.keys()` та `.values()` для виведення всіх ключів і значень.
8. Напишіть програму, яка виводить всі пари «ключ-значення» словника, використовуючи метод `.items()`.
9. Створіть словник з кількома парами «ключ-значення» і використайте метод `.clear()` для його очищення.
10. Напишіть програму, яка створює копію словника за допомогою методу `.copy()`.
11. Створіть два словники та об'єднайте їх за допомогою методу `.update()`.
12. Напишіть програму, яка створює словник, використовуючи два списки: один зі значеннями ключів, інший – зі значеннями.
13. Напишіть програму, яка використовує `collections.defaultdict` для створення словника з дефолтним значенням.

14. Створіть словник і замініть значення для всіх ключів на нове значення за допомогою циклу.

15. Напишіть програму, яка фільтрує словник, залишаючи тільки ті пари ключ-значення, де значення більше заданого порога.

Тема 13
Функції в Python

13.1. Основні поняття.

13.2. Функції зворотного виклику.

13.3. Змінне число параметрів функції.

13.4. Комбінування параметрів.

13.5. Аргументи функції за ключами.

13.6. Значення аргументів функції за замовчуванням.

13.7. Документування функцій.

13.8. Необов'язкові параметри функції та їх зіставлення за ключами.

13.1. Основні поняття

У Python функція визначається як фрагмент коду, який можна викликати (або «викликати») для виконання певної задачі або операції. Визначення функції містить блок коду, який виконується, коли функція викликається, а також може містити параметри, які можна передати у функцію для обробки.

Основні принципи визначення функцій у Python:

1. Ключове слово *def*. Визначення функції починається з ключового слова *def*, за яким слідує ім'я функції та круглі дужки. Наприклад: *def my_function():* .

2. Блок коду. Після заголовка функції слідує блок коду, який виконується під час виклику функції. Цей блок коду відділяється від заголовка функції за допомогою відступів.

3. Параметри (необов'язково). Функція може приймати параметри, які можна використовувати всередині функції для обробки даних. Параметри вказуються у круглих дужках після імені функції. Наприклад: *def my_function(parameter1, parameter2):* .

4. Повернення значення (необов’язково). Функція може повертати значення за допомогою ключового слова *return*. Це значення видається з функції, коли вона завершує своє виконання.

```
def greet(name):
    """Функція для привітання користувача"""
    print("Привіт, " + name + "!")
```

У цьому прикладі *greet* – ім’я функції, яка приймає один параметр *name*. Під час виклику функції *greet('Василь')* вона виведе *Привіт, Василь!* у консоль.

Функції в Python дають змогу створювати загальні блоки коду, які можна використовувати багаторазово у програмі. Це сприяє полегшенню розробки, зменшенню дублювання коду та поліпшенню читабельності.

Інструкція *return*. Інструкція *return* у Python використовується для повернення значення з функції. Вона дає змогу передавати обчислені дані або результати назад до того місця, де була викликана функція.

Основні властивості:

1. Повернення значення. Коли викликається *return*, він повертає вказане значення назад до місця виклику функції.

2. Завершення виконання функції. Виконання функції завершується під час виклику *return*, і будь-який код після нього в тілі функції не виконується.

3. Необов’язкове використання. Використання *return* не є обов’язковим у функціях. Якщо інструкція *return* відсутня, функція завершує своє виконання після виконання всіх інструкцій.

4. Повернення багатьох значень. *return* може повертати багато значень, розділених комами. Ці значення можуть бути упаковані в кортеж або інший тип даних, і можуть бути розпаковані під час виклику функції.

```
def add(a, b):
    """Функція для додавання двох чисел"""
    return a + b

result = add(3, 4)
print(result)  # Виведе: 7

def even_or_odd(num):
    """Функція, яка перевіряє, чи є число парним"""
    if num % 2 == 0:
        return "Парне"
    else:
        return "Непарне"

print(even_or_odd(5))  # Виведе: Непарне
```

У першому прикладі функція *add* повертає суму двох чисел `*a*` і `*b*`. У другому прикладі функція *even_or_odd* повертає рядок, що вказує, чи є задане число парним чи непарним.

13.2. Функції зворотного виклику

Функції зворотного виклику, або *callback*-функції, є функціями, які передаються як аргументи в іншу функцію. Вони використовуються для передачі логіки виконання або дій у функцію, яка буде виконувати певний процес. У Python цей механізм дає змогу створювати більш гнучкі та зручні архітектурні рішення.

Основні особливості функцій зворотного виклику:

1. Передача функції як аргументу. Функція може бути передана в іншу функцію як аргумент.

2. Виконання функції зворотного виклику. У викликаній функції може бути викликана передана функція зворотного виклику.

3. Гнучкість програми. Цей механізм дає змогу розширювати функціональність програми та реагувати на різні події або умови.

```
def apply_operation(x, y, operation):
    """Функція, яка виконує певну операцію над двома числами"""
    result = operation(x, y)
    print("Результат операції:", result)

def add(a, b):
    """Функція додавання"""
    return a + b

def multiply(a, b):
    """Функція множення"""
    return a * b

# Виклик функції apply_operation з функцією add як зворотного виклику
apply_operation(3, 4, add)  # Виведе: Результат операції: 7

# Виклик функції apply_operation з функцією multiply як зворотного виклику
apply_operation(3, 4, multiply)  # Виведе: Результат операції: 12
```

У цьому прикладі функція *apply_operation* приймає два числа `*x*` та `*y*`, а також функцію *operation* як третього аргументу. Потім вона викликає передану функцію *operation* з аргументами `*x*` та `*y*`, обчислює результат і виводить його на екран. Функції *add* та *multiply* використовуються як функції зворотного виклику для виконання операцій додавання та множення відповідно.

13.3. Змінне число параметрів функції

У Python ви можете створювати функції зі змінною кількістю параметрів за допомогою * або у визначенні функції. Це дає змогу створювати більш гнучкі функції, які можуть приймати будь-яку кількість аргументів.

Основні особливості функцій змінної кількості параметрів:

1. Аргументи змінної кількості. Можна визначити функцію так, щоб вона приймала будь-яку кількість аргументів.

2. **args* – використовується для передачі змінної кількості позиційних аргументів у вигляді кортежу.

3. ***kwargs* – використовується для передачі змінної кількості ключових аргументів у вигляді словника.

```
def sum_values(*args):
    """Функція для сумування всіх переданих значень."""
    total = 0
    for num in args:
        total += num
    return total

result = sum_values(1, 2, 3, 4, 5)
print(result)  # Виведе: 15

def print_values(**kwargs):
    """Функція для виведення всіх переданих ключів та значень."""
    for key, value in kwargs.items():
        print(key + ':', value)

print_values(name='John', age=30, city='New York')
# Виведе:
# name: John
# age: 30
# city: New York
```

У першому прикладі функція *sum_values* приймає змінну кількість позиційних аргументів і сумує їх. У другому прикладі функція *print_values* приймає змінну кількість ключових аргументів і виводить їхні пари ключ-значення.

13.4. Комбінування параметрів

У Python ви можете комбінувати різні типи параметрів у визначенні функції, такі як позиційні аргументи, аргументи за замовчуванням, аргументи змінної довжини *(*args)* та аргументи з ключовими значеннями *(**kwargs)*. Це дає змогу створювати дуже гнучкі та розширювані функції.

Основні особливості комбінування параметрів функції:

1. Позиційні аргументи. Аргументи, які передаються функції в порядку, в якому вони визначені у визначенні функції.

2. Аргументи за замовчуванням. Аргументи, які мають значення за замовчуванням і можуть бути опущені під час виклику функції.

3. Змінна кількість параметрів (*args та **kwargs). Параметри, які можуть приймати будь-яку кількість позиційних або ключових аргументів.

4. Порядок аргументів. Порядок визначення параметрів у визначенні функції має значення. Зазвичай позиційні аргументи йдуть першими, після них йдуть аргументи за замовчуванням, а потім – параметри змінної довжини.

```
def example_func(a, b, c=10, *args, **kwargs):
    """Приклад комбінування параметрів функції."""
    print("a:", a)
    print("b:", b)
    print("c:", c)
    print("args:", args)
    print("kwargs:", kwargs)
```

```
example_func(1, 2)  # Виведе:
# a: 1
# b: 2
# c: 10
# args: ()
# kwargs: {}

example_func(1, 2, 3, 4, 5, name='John', age=30)  # Виведе:
# a: 1
# b: 2
# c: 3
# args: (4, 5)
# kwargs: {'name': 'John', 'age': 30}
```

У цьому прикладі функція *example_func* має позиційні аргументи `*a*` і `*b*`, а також аргумент за замовчуванням `*c*`. Після них йдуть параметри змінної довжини **args* та ***kwargs*, які приймають додаткові аргументи без обмежень.

13.5. Аргументи функції за ключами

Аргументи функції за ключами (*keyword arguments*) – це аргументи, які передаються у функцію у вигляді пар ключ-значення. Вони дають змогу передавати аргументи в будь-якому порядку та з заданими назвами, що полегшує розуміння та використання функцій.

Основні особливості аргументів функції за ключами:

1. Задані назви. Аргументи передаються за допомогою їхніх назв (ключів), тому порядок їх подання не має значення.

2. Гнучкість. Аргументи за ключами можуть бути задані окремо від позиційних аргументів, що дає змогу змінювати лише певні параметри функції.

3. Читабельність. Використання аргументів за ключами полегшує зрозуміння, які значення передаються у функцію, особливо коли вона має багато параметрів.

```
def greet(name, message):
    """Функція для виведення привітання."""
    print(f"Привіт, {name}! {message}")

# Виклик функції з аргументами за ключами
greet(name="John", message="Як справи?")  # Виведе: Привіт, John! Як справи?

# Аргументи можуть бути передані в будь-якому порядку
greet(message="Як справи?", name="Alice")  # Виведе: Привіт, Alice! Як справи?
```

У цьому прикладі функція *greet* приймає два аргументи: *name* та *message*. Під час виклику функції аргументи передаються з указанням їхніх назв (ключів). Це дає змогу змінювати порядок аргументів та полегшує зрозуміння, які значення передаються у функцію.

13.6. Значення аргументів функції за замовчуванням

Значення аргументів функції за замовчуванням – це значення, яке функція приймає, якщо аргумент не був переданий під час виклику. Це дає змогу створювати більш гнучкі функції, які можуть працювати з різними наборами параметрів, не вимагаючи від користувача вказувати всі аргументи.

Основні особливості значень аргументів функції за замовчуванням:

1. Зручність. Значення за замовчуванням дають змогу підтримувати стандартні налаштування функцій без необхідності передачі аргументів у кожному виклику.

2. Гнучкість. Користувач може перевизначити значення за замовчуванням, якщо вони не відповідають його потребам.

3. Читабельність. Значення за замовчуванням полегшують розуміння функції та її параметрів.

```
def greet(name='World'):
    """Функція для виведення привітання."""
    print(f"Hello, {name}!")

# Виклики функції з різними аргументами
greet()     # Виведе: Hello, World!
greet('John')# Виведе: Hello, John!
```

У цьому прикладі функція *greet* приймає аргумент *name*, який має значення за замовчуванням *'World'*. Якщо аргумент не передається під час виклику, використовується значення за замовчуванням. В обох випадках виводиться *"Hello, "* разом з ім'ям, яке передається або значенням за замовчуванням, якщо аргумент не вказаний.

13.7. Документування функцій

Документування функцій у Python – це процес написання пояснювальних коментарів для функцій, які пояснюють їхню поведінку, параметри та значення, які вони приймають і повертають. Це робить код більш зрозумілим для інших програмістів, які можуть використовувати ваші функції, а також полегшує роботу з власним кодом у майбутньому.

Основні особливості документування функцій:

1. *Docstrings*. Документація функцій у Python зазвичай реалізується за допомогою рядків-коментарів, які розміщуються безпосередньо після оголошення функції.

2. *PEP 257*. Рекомендації PEP 257 надають стандартний формат для написання документації функцій у Python.

3. Зрозумілість і повнота. Документація повинна бути зрозумілою та повною, щоб інші програмісти змогли легко розібратися у функції без додаткових пояснень.

```
def add(a, b):
    """Функція, що додає два числа.

    Приймає два числа a та b і повертає їхню суму.

    Args:
        a (int): Перше число.
        b (int): Друге число.

    Returns:
        int: Сума двох чисел.

    Example:
        >>> add(2, 3)
        5
    """
    return a + b

def greet(name):
    """Функція, що виводить привітання.

    Приймає ім'я і виводить привітання.

    Args:
        name (str): Ім'я особи, якій потрібно передати привітання.

    Returns:
        None
    """
    print(f"Hello, {name}!")

# Виведення документації функції
print(add.__doc__)
print(greet.__doc__)
```

У цих прикладах функції *add* та *greet* документовані за допомогою *docstrings*. Коментарі розміщуються в межах трьох подвійних лапок та надають інформацію про призначення функції, її параметри, значення, які вона повертає та приклади використання. Це дає змогу іншим програмістам швидко зрозуміти, як працює функція та як її використовувати.

Атрибути функції, вивід списку атрибутів за допомогою функції *dir()*. Атрибути функції – це об'єкти, які пов'язані з функціями у Python і надають інформацію про їх властивості та функціональність. Метод *dir()* використовується для виведення списку атрибутів об'єкта, охоплюючи атрибути функцій. Для функцій він виводить список доступних атрибутів, які можна використовувати для взаємодії з функціями.

Основні атрибути функцій, які можна отримати за допомогою методу *dir()*, охоплюють:

1. `__*doc*__` – документація функції, якщо вона була задокументована за допомогою *docstrings*.
2. `__*name*__` – ім'я функції.
3. `__*defaults*__` – кортеж, що містить значення за замовчуванням для аргументів функції.
4. `__*code*__` – об'єкт, який містить скомпільований об'єкт коду функції.
5. `__*globals*__` – словник, що містить глобальні змінні функції.
6. `__*annotations*__` – словник, що містить анотації функції.

```
def greet(name):
    """Функція, що виводить привітання."""
    print(f"Hello, {name}!")

# Виведення атрибутів функції за допомогою dir()
print(dir(greet))
```

Цей код виведе список атрибутів функції *greet*, охоплюючи __*doc*__, __*name*__ та інші атрибути, які можна використовувати для отримання інформації про функцію та її властивості.

Строгий порядок слідування визначення та виклику функції. У Python існує строгий порядок визначення та виклику функцій, який важливий для правильної роботи програми. Ось деякі важливі аспекти цього порядку:

1. Визначення функцій перед викликом. Функції повинні бути визначені до їх виклику. Це означає, що код, який визначає функції, повинен розташовуватися перед кодом, який їх викликає. Якщо ви намагаєтеся викликати функцію, яка не була визначена, ви отримаєте помилку.

2. Виклик функції з правильним ім'ям. Під час виклику функції потрібно вказати її ім'я разом з обов'язковими або необов'язковими аргументами, якщо вони є. Необхідно використовувати правильне ім'я функції, інакше програма не знатиме, яку функцію викликати.

3. Параметри функції передаються в правильному порядку. Якщо функція приймає аргументи, вони повинні бути передані в правильному порядку. Порядок передачі аргументів повинен відповідати порядку, в якому вони визначені у визначенні функції.

4. Виклик функції з правильною кількістю аргументів. Якщо функція визначається з певною кількістю параметрів, під час її виклику необхідно передати правильну кількість аргументів.

5. Правильне використання зворотного значення. Якщо функція повертає значення, це значення повинно бути правильно використано у програмі.

Невиконання будь-якого з цих правил може призвести до помилки виконання програми або неправильного результату. Тому важливо дотримуватися строгого порядку визначення та виклику функцій у Python.

13.8. Необов'язкові параметри функції та їх зіставлення за ключами

У Python можна використовувати необов'язкові параметри функцій, які мають значення за замовчуванням. Це дає змогу задавати аргументи функції за замовчуванням, які можна не передавати під час виклику функції. Також можна використовувати

зіставлення аргументів за ключами, що дає змогу передавати аргументи в будь-якому порядку.

```
def greet(name, message="Hello"):
    """Функція, що виводить повідомлення з привітанням.
    Args:
        name (str): Ім'я особи.
        message (str, optional): Повідомлення. За замовчуванням "Hello".
    """
    print(f"{message}, {name}!")

# Виклик функції з обов'язковим та необов'язковим аргументами
greet("Alice")  # Виведе: Hello, Alice!
greet("Bob", "Good morning")  # Виведе: Good morning, Bob!

# Виклик функції зі зіставленням аргументів за ключами
greet(message="Hi", name="Charlie")  # Виведе: Hi, Charlie!
```

У цьому прикладі функція *greet* має обов'язковий параметр *name* та необов'язковий параметр *message*, який має значення за замовчуванням *"Hello"*. Під час виклику функції можна передавати обидва аргументи або тільки один з них. Крім того, використано зіставлення аргументів за ключами, що дає змогу передавати аргументи в будь-якому порядку.

Перелік питань для самоконтролю

1. Визначення функції як фрагменту коду багаторазового використання.
2. Схема створення функції за допомогою ключового слова **def.**
3. Інструкція **return.**

4. Функції зворотного виклику.

5. Комбінування параметрів.

6. Атрибути функції, вивід списку атрибутів за допомогою функції **dir().**

7. Строгий порядок слідування визначення та виклику функції.

8. Необов’язкові параметри функції та їх зіставлення за ключами.

Тестові питання

1. Що таке функція в Python:

а) змінна, яка зберігає значення;

б) блок коду, який можна викликати для виконання певної задачі;

в) клас, який визначає поведінку об’єкта;

г) спеціальний тип даних для зберігання послідовності об’єктів.

2. Яка основна перевага використання функцій у програмуванні:

а) зменшення кількості рядків коду;

б) покращення продуктивності розробки;

в) можливість створення об’єктів;

г) легше зберігання даних.

3. Як визначається функція в Python:

а) за допомогою ключового слова `def`;

б) за допомогою ключового слова `func`;

в) за допомогою ключового слова `function`;

г) за допомогою ключового слова `define`.

4. Який синтаксис використовується для визначення параметрів функції в Python:

а) параметри вказуються у круглих дужках, розділені комами;

б) параметри вказуються у фігурних дужках, розділені комами;

в) параметри вказуються у квадратних дужках, розділені комами;

г) параметри вказуються у трикутних дужках, розділені комами.

5. Яка операція виконується для виклику функції в Python:

а) `execute()`;

б) `run()`;

в) `call()`;

г) `invoke()`.

6. Що таке «повернення значення» у функції Python:

а) процес завершення виконання функції;

б) процес повернення курсору до початку функції;

в) видача результату функції для подальшого використання;

г) очікування введення користувача.

7. Яке ключове слово використовується для виходу з функції передчасно:

а) `exit`;

б) `break`;

в) `stop`;

г) `return`.

8. Що таке аргументи функції:

а) значення, які функція повертає;

б) значення, які передаються функції під час виклику;

в) значення, які зберігаються у функції;

г) імена параметрів функції.

9. Яка функція використовується для друку значення в консоль у Python:

а) `print()`;

б) `write()`;

в) `display()`;

г) `show()`.

10. Що таке рекурсивна функція:

а) функція, що викликається з консолі;

б) функція, що викликає сама себе;

в) функція, яка працює тільки з рядками;

г) функція, яка викликається тільки один раз.

Рекомендовані джерела:

Основні: 1; 2; 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть функцію `get_number()`, яка повертає число 5. Викличте функцію і виведіть результат.

2. Створіть функцію `add(a, b)`, яка приймає два параметри і повертає їхню суму. Викличте цю функцію з двома числами.

3. Напишіть функцію `greet()`, яка виводить на екран повідомлення "Hello, World!". Викличте цю функцію.

4. Створіть функцію `multiply(x, y, z)`, яка приймає три параметри і повертає їхній добуток. Викличте цю функцію з трьома числами.

5. Напишіть функцію `power(base, exponent=2)`, яка підносить число `base` до степеня `exponent` (за замовчуванням 2). Викличте функцію без другого параметра.

6. Напишіть функцію `print_info(name, *args)`, яка виводить ім’я і всі додаткові аргументи. Викличте функцію з ім’ям та кількома додатковими параметрами.

7. Створіть функцію `display_info(name, **kwargs)`, яка виводить ім’я і всі додаткові ключові аргументи. Викличте функцію з ім’ям та кількома ключовими аргументами.

8. Напишіть функцію `min_max(numbers)`, яка приймає список чисел і повертає кортеж з мінімальним і максимальним значеннями.

9. Створіть функцію `is_even(number)`, яка повертає `True`, якщо число парне, і `False`, якщо непарне. Викличте цю функцію з кількома числами.

10. Напишіть функцію `contains_substring(string, substring)`, яка перевіряє, чи є підрядок `substring` в рядку `string`. Викличте функцію з рядками.

11. Напишіть рекурсивну функцію `factorial(n)`, яка обчислює факторіал числа `n`.

12. Створіть рекурсивну функцію `fibonacci(n)`, яка обчислює `n`-е число Фібоначчі.

13. Напишіть функцію `update_count()`, яка збільшує глобальну змінну `count` на 1.

14. Створіть функцію `safe_divide(a, b)`, яка ділить `a` на `b`, але обробляє виключення поділу на нуль.

15. Напишіть функцію `is_palindrome(string)`, яка перевіряє, чи є рядок паліндромом.

Тема 14
Анонімні функції та декоратори. Видимість глобальних та локальних змінних

14.1. Анонімні функції lambda.

14.2. Декоратори функцій.

14.3. Видимість глобальних та локальних змінних.

14.4. Рекурсія.

14.5. Вкладені функції. Атрибут об’єкта функції __anotations__.

14.6. Замикання в Python.

14.1. Анонімні функції lambda

Анонімні функції, відомі також як *lambda*-функції, є функціями без імені. Вони використовуються для створення коротких функцій, які можуть бути визначені та викликані на місці.

Призначення. По-перше, створення коротких функцій без необхідності додавання додаткових рядків коду для визначення окремої функції за допомогою ключового слова *def*. По-друге, використання в місцях, де потрібна невелика функція, наприклад, у функціях вищого порядку, які очікують аргументу-функції.

Детальний опис:

1. Анонімні функції визначаються за допомогою ключового слова *lambda*, а не *def*.

2. Вони можуть мати будь-яку кількість аргументів, але можуть містити тільки один вираз.

3. Результат виразу стає значенням повернення функції.

4. Після визначення анонімні функції можуть бути викликані як будь-яка інша функція.

```
# lambda-функція, яка додає два числа
add = lambda x, y: x + y

# lambda-функція, яка повертає квадрат аргументу
square = lambda x: x  2

# Виклик lambda-функції для додавання чисел
result = add(3, 5)  # Результат буде 8

# Виклик lambda-функції для обчислення квадрата числа
square_result = square(4)  # Результат буде 16
```

У цьому прикладі *lambda*-функції використовуються для створення коротких функцій без необхідності визначення їх окремо за допомогою *def*, а потім вони викликаються для виконання певних операцій.

Способи задавання та сфери використання анонімних функцій.

Анонімні (*lambda*) функції в Python можуть бути корисними в різних ситуаціях, де потрібно визначити коротку функцію, або коли функція використовується лише один раз.

1. Сортування за допомогою вбудованої функції *sorted* з використанням *lambda*-функції:

```
# Сортування списку слов'ями за довжиною слів
words = ["apple", "banana", "cherry", "date"]
sorted_words = sorted(words, key=lambda x: len(x))
print(sorted_words)  # Виведе: ['date', 'apple', 'banana', 'cherry']
```

2. Фільтрація списку за допомогою вбудованої функції *filter* та *lambda*-функції:

```
# Відфільтрувати список чисел, залишивши тільки парні
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Виведе: [2, 4, 6, 8, 10]
```

3. Мапування значень у список за допомогою вбудованої функції *map* та *lambda*-функції:

```
# Піднести кожне число у списку до квадрата
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x  2, numbers))
print(squared_numbers)  # Виведе: [1, 4, 9, 16, 25]
```

4. Створення словника за допомогою *lambda*-функції виразу:

```
# Створити словник,
#  де ключ – це кожен елемент зі списку, а значення – його квадрат
numbers = [1, 2, 3, 4]
squared_dict = {x: x  2 for x in numbers}
print(squared_dict)  # Виведе: {1: 1, 2: 4, 3: 9, 4: 16}
```

5. Використання виключень за допомогою *lambda*-функції:

```
# Вибрати перші дві цифри кожного числа у списку, якщо вони доступні
numbers = [123, 45, 6, 789]
first_two_digits = list(map(lambda x: int(str(x)[:2]), numbers))
print(first_two_digits)  # Виведе: [12, 45, 6, 78]
```

6. Сортування словника за значеннями за допомогою *lambda*-функції:

```
# Сортування словника за значеннями у спадаючому порядку
my_dict = {'a': 10, 'b': 5, 'c': 20, 'd': 15}
sorted_dict = dict(sorted(my_dict.items(), key=lambda x: x[1], reverse=True))
print(sorted_dict)  # Виведе: {'c': 20, 'd': 15, 'a': 10, 'b': 5}
```

Ці приклади показують різноманітні сфери використання *lambda*-функцій у Python, включаючи фільтрацію, сортування, мапування та обробку даних.

14.2. Декоратори функцій

Декоратори – це спосіб модифікувати функціональність існуючої функції в Python. Вони дають змогу обгортати функцію, щоб змінити або розширити її поведінку.

1. Простий декоратор для виведення повідомлення до та після виклику функції:

```
def my_decorator(func):
    def wrapper():
        print("Початок виконання функції")
        func()
        print("Завершення виконання функції")
    return wrapper

@my_decorator
def say_hello():
    print("Привіт, світ!")

say_hello()
```

2. Декоратор з аргументами:

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, kwargs):
            for _ in range(n):
                func(*args, kwargs)
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Привіт, {name}!")

greet("Саша")
```

3. Декоратор для перевірки аргументів функції:

```
def validate(func):
    def wrapper(*args, kwargs):
        for arg in args:
            if not isinstance(arg, int):
                raise TypeError("Аргументи мають бути цілими числами")
        return func(*args, kwargs)
    return wrapper

@validate
def add(x, y):
    return x + y

print(add(3, 5))  # Виведе: 8
```

4. Декоратор для вимірювання часу виконання функції:

```
import time

def timer(func):
    def wrapper(*args, kwargs):
        start_time = time.time()
        result = func(*args, kwargs)
        end_time = time.time()
        print(f"Час виконання {func.__name__}: {end_time - start_time} секунд")
        return result
    return wrapper

@timer
def some_function():
    time.sleep(2)

some_function()
```

5. Декоратор для кешування результатів функції:

```
def memoize(func):
    cache = {}

    def wrapper(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result
    return wrapper

@memoize
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(10))  # Виведе: 55
```

6. Декоратор для логування:

```
def logger(func):
    def wrapper(*args, kwargs):
        print(f"Виклик функції {func.__name__} з аргументами {args},
{kwargs}")
        result = func(*args, kwargs)
        print(f"Результат: {result}")
        return result
    return wrapper

@logger
def multiply(x, y):
    return x * y

print(multiply(3, 5))  # Виведе: 15
```

7. Декоратор для заборони виклику функції в певний період часу:

```
import time
import functools

def timeout(seconds):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, kwargs):
            start_time = time.time()
            result = func(*args, kwargs)
            end_time = time.time()
            elapsed_time = end_time - start_time
            if elapsed_time > seconds:
                raise TimeoutError(f"Час виконання перевищив {seconds} секунд")
            return result
        return wrapper
    return decorator

@timeout(3)
def slow_function():
    time.sleep(5)

slow_function()  # Викличе помилку TimeoutError
```

Ці приклади демонструють різні сценарії застосування декораторів для зміни поведінки функцій. Вони дають змогу розширити функціональність без зміни коду функцій, що збільшує їхню гнучкість та повторне використання.

14.3. Видимість глобальних та локальних змінних

У Python видимість змінних може бути глобальною або локальною. Локальні змінні доступні лише в межах блоку коду, де вони були оголошені, тоді як глобальні змінні доступні з будь-якого місця у програмі.

1. Локальні та глобальні змінні в функції:

```
global_var = 10

def my_function():
    local_var = 5
    print("Локальна змінна:", local_var)
    print("Глобальна змінна:", global_var)

my_function()
print("Глобальна змінна поза функцією:", global_var)
# print(local_var)  # Помилка: NameError: name 'local_var' is not defined
```

2. Зміна значення глобальної змінної всередині функції:

```
x = 10

def change_global():
    global x
    x = 20

change_global()
print(x)  # Виведе: 20
```

3. Локальна змінна, що перекриває глобальну змінну:

```
x = 10

def my_function():
    x = 5
    print("Значення x у функції:", x)

my_function()
print("Значення x поза функцією:", x)  # Виведе: 10
```

4. Локальна змінна з тією самою назвою, що й глобальна, без зміни глобальної змінної:

```
x = 10

def my_function():
    x = x + 5   # Помилка: UnboundLocalError: local variable 'x' referenced before assignment
    print("Значення x у функції:", x)

my_function()
```

5. Глобальна змінна, змінена у функції:

```
x = 10

def my_function():
    global x
    x = x + 5
    print("Значення x у функції:", x)

my_function()
print("Значення x поза функцією:", x)  # Виведе: 15
```

6. Використання ключового слова *nonlocal* для зміни значення змінної в об'єкті, що оточує функцію:

```
def outer_function():
    x = 10

    def inner_function():
        nonlocal x
        x = 20

    inner_function()
    print("Значення x у функції:", x)

outer_function()  # Виведе: 20
```

7. Локалізація змінної в циклі:

```
for i in range(3):
    var = i
print(var)  # Виведе: 2
```

Ці приклади ілюструють різні ситуації, пов'язані з локальними та глобальними змінними, а також використання ключових слів *global* та *nonlocal* для роботи з ними.

Одержання словників глобальних та локальних ідентифікаторів за допомогою функцій *globals()* та *locals()*.

Функції *globals()* та *locals()* в Python дають змогу отримувати доступ до словників глобальних та локальних ідентифікаторів відповідно.

1. *globals():*

```
x = 10
y = 20

def my_function():
    z = 30
    print("Глобальні змінні:", globals())

my_function()
```

У цьому прикладі *globals()* поверне словник з усіма глобальними змінними та їхніми значеннями, включаючи `x` і `y`.

2. *locals():*

```
x = 10
y = 20

def my_function():
    z = 30
    print("Локальні змінні:", locals())

my_function()
```

У цьому прикладі *locals()* поверне словник з усіма локальними змінними та їхніми значеннями в межах функції *my_function()*, включаючи `z`.

Зверніть увагу, що ці функції повертають словник, тому їх можна використовувати для динамічного доступу до змінних у програмі. Однак це може бути корисно лише для налагодження або динамічного створення коду. Використання цих функцій у виробничому коді може зробити код менш зрозумілим і таким, що викликатиме помилки.

14.4. Рекурсія

Рекурсія – це техніка в програмуванні, коли функція викликає саму себе для обробки певного завдання. Ось декілька прикладів рекурсивних функцій у Python:

1. Обчислення факторіала:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
print("Факторіал числа 5:", result)  # Виведе: 120
```

2. Сума чисел у списку:

```
def sum_list(lst):
    if not lst:
        return 0
    else:
        return lst[0] + sum_list(lst[1:])

numbers = [1, 2, 3, 4, 5]
result = sum_list(numbers)
print("Сума чисел у списку:", result)  # Виведе: 15
```

3. Обчислення числа Фібоначчі:

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

result = fibonacci(6)
print("Шосте число Фібоначчі:", result)  # Виведе: 8
```

4. Підрахунок кількості елементів у списку:

```
def count_elements(lst):
    if not lst:
        return 0
    else:
        return 1 + count_elements(lst[1:])

numbers = [1, 2, 3, 4, 5, 6, 7]
result = count_elements(numbers)
print("Кількість елементів у списку:", result)  # Виведе: 7
```

14.5. Вкладені функції. Атрибут об’єкта функції __anotations__

Вкладені функції – це функції, що оголошені всередині інших функцій. Вони можуть бути корисними для організації коду та зменшення повторення коду. Анотації функцій – це спеціальні мітки або метадані, що можуть бути додані до параметрів та значень, щоб вказати їхні типи, призначення та інші деталі.

```
def outer_function(x: int) -> int:
    def inner_function(y: int) -> int:
        return y * 2

    result = inner_function(x)
    return result

print(outer_function(5))  # Виведе: 10

# Анотації функцій можуть бути отримані через атрибут __annotations__
print(outer_function.__annotations__)  # Виведе: {'x': <class 'int'>, 'return': <class 'int'>}
```

У цьому прикладі *inner_function* є вкладеною функцією, оголошеною всередині *outer_function*. Параметри та значення, що повертаються обидвома функціями, мають анотації типів (у цьому випадку – цілі числа), які можна отримати через атрибут *__annotations__* об’єкта функції.

14.6. Замикання в Python

1. Замикання для підрахунку суми:

```
def outer_function(x):
    def inner_function(y):
        return x + y

    return inner_function

closure = outer_function(5)
print(closure(3))  # Виведе: 8
```

2. Замикання для обчислення середнього значення:

```
def average():
    numbers = []

    def add_number(number):
        numbers.append(number)
        return sum(numbers) / len(numbers)

    return add_number

calculate_average = average()
print(calculate_average(10))  # Виведе: 10.0
print(calculate_average(20))  # Виведе: 15.0
print(calculate_average(30))  # Виведе: 20.0
```

3. Замикання для генерації послідовності чисел:

```
    def sequence(start):
        counter = start
```

```
    def next_number():
        nonlocal counter
        result = counter
        counter += 1
        return result

    return next_number

get_next = sequence(10)
print(get_next())  # Виведе: 10
print(get_next())  # Виведе: 11
print(get_next())  # Виведе: 12
```

У цих прикладах замикання дають змогу створювати функції, які зберігають стан (наприклад, поточне значення змінної *counter* в прикладі 3) та можуть використовувати його пізніше під час виклику функції.

Перелік питань для самоконтролю

1. Анонімні функції **lambda.**
2. Способи задавання та сфери використання анонімних функцій.
3. Декоратори функцій.
4. Видимість глобальних та локальних змінних.
5. Одержання словників глобальних та локальних ідентифікаторів за допомогою функцій **globals()** та **locals().**
6. Рекурсія.
7. Вкладені функції.
8. Анотації функцій.
9. Атрибут об’єкта функції **__anotations__.**

Тестові питання

1. Що таке анонімна функція в Python:

а) функція, яка має анонімний параметр;

б) функція, яка не має імені;

в) функція, яка приховує свій вміст;

г) функція, яка не приймає жодних параметрів.

2. Яким ключовим словом визначається анонімна функція в Python:

а) `def`;

б) `function`;

в) `lambda`;

г) `anon`.

3. Який оператор використовується для виклику анонімної функції в Python:

а) `call`;

б) `invoke`;

в) `run`;

г) `()` (дужки).

4. Що робить функція `lambda` в Python:

а) створює анонімну функцію;

б) збільшує значення аргумента на 1;

в) зменшує значення аргумента на 1;

г) перевіряє, чи є аргумент парним числом.

5. Що таке декоратор у Python:

а) функція, яка додає підпис до коду;

б) функція, яка перетворює аргументи;

в) функція, яка робить зміни в іншій функції;

г) функція, яка виводить додаткову інформацію під час виконання програми.

6. Яке ключове слово використовується для застосування декоратора до функції в Python:

а) `apply`;
б) `decorate`;
в) `decorator`;
г) `@`.

7. Що робить декоратор `@staticmethod` у Python:

а) робить функцію статичною;
б) додає додаткову властивість до функції;
в) робить функцію абстрактною;
г) забороняє виклик функції ззовні.

8. Яке призначення декоратора `@property` в Python:

а) додає властивість до класу;
б) забезпечує доступ до атрибута як до властивості;
в) додає статичний метод до класу;
г) додає метод до класу для отримання значення властивості.

9. Яке ключове слово використовується для визначення власного декоратора в Python:

а) `decorator`;
б) `create`;
в) `define`;
г) `def`.

10. Що робить декоратор `@wraps` у Python:

а) замінює декоровану функцію новою функцією;
б) забезпечує декоратору доступ до змінних функції;
в) додає додатковий аргумент до функції;
г) зберігає метадані оригінальної функції.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть lambda-функцію, яка приймає два числа і повертає їхню суму. Викличте цю функцію і виведіть результат.

2. Створіть lambda-функцію, яка приймає число і повертає його квадрат. Використовуйте цю функцію для обчислення квадрата числа 7.

3. Використовуйте lambda-функцію разом з `filter()`, щоб залишити тільки парні числа в списку.

4. Напишіть lambda-функцію для сортування списку рядків за їхньою довжиною.

5. Створіть lambda-функцію, яка приймає три числа і повертає їхній добуток. Використовуйте цю функцію для обчислення добутку чисел 2, 3 і 4.

6. Напишіть декоратор, який виводить "Function called" перед викликом будь-якої функції.

7. Створіть декоратор, який вимірює і виводить час виконання функції.

8. Напишіть декоратор `repeat`, який повторює виклик функції певну кількість разів, передану як параметр.

9. Створіть декоратор, який перевіряє, чи є аргументи функції правильного типу.

10. Напишіть декоратор, який обробляє будь-які виключення, що виникають під час виконання функції, і виводить повідомлення про помилку.

11. Створіть функцію, яка змінює локальну змінну і виводить її значення. Переконайтеся, що глобальна змінна не змінюється.

12. Напишіть функцію, яка використовує `global` для модифікації глобальної змінної.

13. Створіть функцію з вкладеними функціями, де внутрішня функція змінює змінну з зовнішньої функції за допомогою `nonlocal`.

14. Напишіть декоратор, який змінює значення глобальної змінної під час кожного виклику функції.

15. Створіть lambda-функцію, яка використовує локальні змінні з оточуючого контексту.

Тема 15
Функції-генератори

15.1. Поняття функції-генератора.

15.2. Виклик функції-генератора з функції-генератора за допомогою ключового слова yield.

15.3. Застосування методу __next__ до функцій-генераторів.

15.4. Застосування функції-генератора.

15.1. Поняття функції-генератора

Функції-генератори – це особливий вид функцій у Python, які дають змогу генерувати значення одне за одним за запитом, не завантажуючи всю послідовність у пам’ять.

1. Генерація послідовності чисел:

```
def count_up_to(limit):
    count = 1
    while count <= limit:
        yield count
        count += 1

for num in count_up_to(5):
    print(num)
```

2. Генерація послідовності Фібоначчі:

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib_gen = fibonacci()
for _ in range(10):
    print(next(fib_gen))
```

3. Генерація випадкових чисел:

```
import random

def random_numbers():
    while True:
        yield random.randint(1, 100)

rand_gen = random_numbers()
for _ in range(5):
    print(next(rand_gen))
```

У цих прикладах функції-генератори використовуються для генерації послідовностей чисел (наприклад, чисел від 1 до певного ліміту, чисел Фібоначчі) або випадкових значень. Вони працюють ефективно, оскільки генерують значення лише за запитом, а не завантажують всю послідовність у пам’ять одразу.

15.2. Виклик функції-генератора з функції-генератора за допомогою ключового слова yield

Оператор *yield* у Python використовується для створення функцій-генераторів. Він дає змогу функції повертати значення, а потім продовжувати виконання з того місця, де було викликано *yield*, коли функція була викликана знову.

Принцип роботи *yield*:

1. Коли функція викликається і виконується, вона виконується до першого виклику оператора *yield*.

2. Функція призупиняє виконання і повертає значення, яке вказано після *yield*.

3. Під час наступного виклику функції вона продовжить виконання з того місця, де було зупинено, і відновить свій стан.

1. Генерація послідовності чисел:

```
def count_up_to(limit):
    count = 1
    while count <= limit:
        yield count
        count += 1

for num in count_up_to(5):
    print(num)
```

2. Генерація Фібоначчі:

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib_gen = fibonacci()
for _ in range(10):
    print(next(fib_gen))
```

3. Генерація випадкових чисел:

```
import random

def random_numbers():
    while True:
        yield random.randint(1, 100)

rand_gen = random_numbers()
for _ in range(5):
    print(next(rand_gen))
```

4. Генерація символів строки:

```
def get_characters(text):
    for char in text:
        yield char

text_gen = get_characters("Hello")
for char in text_gen:
    print(char)
```

5. Генерація потоку даних з файлу:

```
def read_file_lines(file_name):
    with open(file_name, 'r') as file:
        for line in file:
            yield line.strip()

for line in read_file_lines('data.txt'):
    print(line)
```

Оператор *`yield`* дає змогу легко створювати функції, які можуть генерувати послідовності значень одне за одним, ефективно використовуючи ресурси пам’яті.

Можна викликати функцію-генератор з іншої функції-генератора за допомогою ключового слова *yield*. Це називається "делегуванням генератора".

```
def generator1():
    yield 1
    yield 2

def generator2():
    yield 'a'
    yield 'b'
    # Виклик функції-генератора generator1 з функції-генератора generator2
    yield from generator1()
```

```
    yield 'c'
    yield 'd'

gen = generator2()
for value in gen:
    print(value)
```

У цьому прикладі функція-генератор *generator2* викликає функцію-генератор *generator1* за допомогою ключового слова *yield from*. Це дає змогу об'єднати послідовність значень з `generator1` зі значеннями, які вона сама генерує. Результатом буде послідовність: *'a', 'b', 1, 2, 'c', 'd'*.

15.3. Застосування методу __next__ до функцій-генераторів

Метод *__next__* є спеціальним методом, який дає змогу отримувати наступний елемент з функцій-генераторів. Цей метод дає змогу керувати виконанням генератора вручну.

```
def count_up_to(limit):
    count = 1
    while count <= limit:
        yield count
        count += 1

# Створення функції-генератора
gen = count_up_to(3)

# Виклик методу __next__ для отримання наступного елемента
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
# При спробі отримати більше елементів, які є, викине StopIteration
print(next(gen))  # StopIteration:
```

Коли всі значення вже були згенеровані, подальші виклики методу *__next__* викличе виняток *StopIteration*, що можна використовувати для визначення кінця послідовності.

Метод *__next__* дає змогу контролювати виконання функцій-генераторів вручну, але для зручності частіше використовують цикл *for*, який автоматично обробляє виняток *StopIteration*.

15.4. Застосування функції-генератора

1. Генерація послідовності чисел:

```
def generate_numbers(n):
    for i in range(n):
        yield i

# Використання:
for num in generate_numbers(5):
    print(num)
```

2. Читання великого файлу партіями:

```
def read_large_file(file_path, batch_size=1000):
    with open(file_path, 'r') as file:
        while True:
            batch = [next(file) for _ in range(batch_size)]
            if not batch:
                break
            yield batch

# Використання:
for batch in read_large_file('large_file.txt'):
    process_batch(batch)
```

3. Генерація нескінченної послідовності значень:

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Використання:
fib_gen = fibonacci()
for _ in range(10):
    print(next(fib_gen))
```

Ці приклади показують різноманітні застосування функцій-генераторів у Python для ефективної роботи з пам’яттю та обробки великих обсягів даних.

Перелік питань для самоконтролю

1. Поняття функції-генератора.
2. Виклик функції-генератора з функції-генератора за допомогою ключового слова **yield.**
3. Застосування методу **__next__** до функцій-генераторів.
4. Застосування функції-генератора.

Тестові питання

1. Що таке функція-генератор у Python:

а) функція, яка створює випадкові числа;

б) функція, яка генерує послідовність значень за запитом;

в) функція, яка визначає генетичні характеристики об’єктів;

г) функція, яка видає список усіх можливих комбінацій заданих значень.

2. Яка основна перевага використання функцій-генераторів:

а) вони можуть генерувати великі обсяги даних у пам’яті;

б) вони можуть працювати швидше, ніж звичайні функції;

в) вони дають змогу створювати послідовність значень безпосередньо в циклі;

г) вони автоматично перевіряють правильність вхідних даних.

3. Яка функція використовується для створення функції-генератора:

а) `def`;

б) `generator`;

в) `generate`;

г) `yield`.

4. Що повертає функція-генератор під час кожного виклику:

а) ціле число;

б) список;

в) генератор;

г) рядок.

5. Який оператор використовується для повернення значення з функції-генератора:

а) `return`;

б) `yield`;

в) `send`;

г) `receive`.

6. Як функція-генератор указує, що потрібно повернути значення:

а) `return`;

б) `yield`;

в) `output`;

г) `produce`.

7. Що робить ключове слово `yield` у функції-генераторі:

а) повертає значення і завершує функцію;

б) повертає значення, але не завершує функцію;

в) повертає послідовність значень одночасно;

г) зупиняє виконання програми.

8. Як можна використовувати функцію-генератор у циклі `for`:

а) присвоїти результат функції змінній;

б) використовувати функцію як параметр;

в) викликати функцію безпосередньо;

г) використовувати `yield` у тілі циклу.

9. Яка властивість функції-генератора дає змогу зупиняти його виконання та відновлювати його з того самого місця:

а) повернення значення;

б) змінна, що запам’ятовує поточне положення;

в) зупинка виконання програми;

г) виклик функції.

10. Що станеться під час виклику функції-генератора без обробки результату:

а) викинеться помилка;

б) програма завершиться;

в) функція буде виконуватися без зупинки;

г) нічого не станеться, якщо результат не зберігається.

Рекомендовані джерела:

Основні: 1; 2; 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть генератор `count_up_to(n)`, який генерує числа від 1 до `n` (включно).

2. Створіть генератор `squares(numbers)`, який генерує квадрати чисел з переданого списку.

3. Напишіть генератор `even_numbers(n)`, який генерує парні числа від 2 до `n`.

4. Створіть генератор `fibonacci()`, який генерує нескінченну послідовність чисел Фібоначчі.

5. Напишіть генератор `combinations(elements, r)`, який генерує всі можливі комбінації з `r` елементів з переданого списку `elements`.

6. Напишіть генератор `powers_of(base, n)`, який генерує степені `base` від 1 до `n`.

7. Створіть генератор `range_generator(start, end)`, який генерує числа в заданому діапазоні `[start, end)`.

8. Напишіть генератор `random_numbers(n, start, end)`, який генерує `n` випадкових чисел у діапазоні від `start` до `end`.

9. Створіть генератор `repeat_string(string, times)`, який генерує рядки, що складаються з повторень `string` `times` разів.

10. Напишіть генератор `word_pairs(words)`, який генерує пари сусідніх слів зі списку `words`.

11. Створіть генератор `words_from_string(string)`, який генерує слова з рядка `string`, розділені пробілами.

12. Напишіть генератор `rotations(elements)`, який генерує всі можливі обертання списку `elements`.

13. Створіть генератор `progressive_decrease(string)`, який генерує рядки, де довжина зменшується на один символ з кожним кроком.

14. Напишіть генератор `substrings(string)`, який генерує всі можливі підрядки рядка `string`.

15. Створіть генератор `primes()`, який генерує нескінченну послідовність простих чисел.

Тема 16
Pattern matching

16.1. Конструкція match.

16.2. Кортежі в pattern matching.

16.3. Альтернативні значення, пропуск елементів, кортеж із невизначеною кількістю елементів.

16.4. Масиви в pattern matching.

16.5. Масиви невизначеної довжини, альтернативні значення.

16.6. Словники в pattern matching.

16.7. Отримання значень ключів, отримання всіх значень.

16.1. Конструкція match

Конструкція *`match`* уведена в Python 3.10 і використовується для *pattern matching*, що дає змогу здійснювати структурне порівняння даних.

1. Ключове слово *match* – вводить нову конструкцію мови, яка дає змогу виконувати *pattern matching*.

2. Вираз для порівняння – це значення або вираз, який потрібно порівняти з різними шаблонами.

3. *case* – визначає шаблон та відповідний блок коду, який буде виконано, якщо вираз співпадає з цим шаблоном.

4. Шаблони – визначають структуру, за якою вираз буде порівнюватися. Може бути використано різні типи шаблонів, такі як константи, змінні та інші.

5. Значення-заміщувачі (*capture values*) – можливість використовувати іменовані або анонімні значення, які будуть заміщувати частини, що відповідають шаблону.

```
def check_value(value):
    match value:
        case 1:
            print("Значення дорівнює 1")
        case 2:
            print("Значення дорівнює 2")
```

```
        case _:
            print("Значення не відповідає жодному шаблону")

    check_value(2)
```

У цьому прикладі *value* порівнюється з різними шаблонами, і виконується блок коду, що відповідає першому співпадінню, або блок коду після останнього шаблону, якщо жоден із них не співпадає.

16.2. Кортежі в pattern matching

У Python 3.10 з уведенням конструкції *match* можна використовувати кортежі як шаблон для *pattern matching*. Це дає змогу зручно порівнювати структуровані дані, такі як кортежі, з різними шаблонами.

```
# Функція, яка приймає кортеж і виконує pattern matching
def process_tuple(tup):
    match tup:
        case (0, 0):
            print("Кортеж містить два нулі")
        case (x, 0):
            print(f"Кортеж містить {x} та 0")
        case (x, y) if x == y:
            print(f"Кортеж містить два однакові числа: {x}")
        case (x, y):
            print(f"Кортеж містить два різні числа: {x} та {y}")

# Виклик функції з різними кортежами
process_tuple((0, 0))  # Виводить: Кортеж містить два нулі
process_tuple((3, 0))  # Виводить: Кортеж містить 3 та 0
process_tuple((2, 2))  # Виводить: Кортеж містить два однакові числа: 2
process_tuple((1, 5))  # Виводить: Кортеж містить два різні числа: 1 та 5
```

У цьому прикладі *process_tuple* приймає кортеж та порівнює його з різними шаблонами в конструкції *match*. Шаблони можуть містити конкретні значення, змінні та умови. Кожний *case* відповідає певному шаблону кортежа, і відповідний блок коду виконується в разі співпадіння.

16.3. Альтернативні значення, пропуск елементів, кортеж із невизначеною кількістю елементів

У Python 3.10 з уведенням конструкції `match` використання альтернативних значень, пропуску елементів та кортежів з невизначеною кількістю елементів стало зручнішим.

1. Альтернативні значення:

```
def check_value(value):
    match value:
        case 0 | 1 | 2:
            print("Значення належить до множини {0, 1, 2}")
        case 3 | 4 | 5:
            print("Значення належить до множини {3, 4, 5}")
        case _:
            print("Значення не належить жодній із вказаних множин")

check_value(4)  # Виводить: Значення належить до множини {3, 4, 5}
```

2. Пропуск елементів:

```
def process_tuple(tup):
    match tup:
        case (x, _, z):
            print(f"Перший та третій елементи кортежу: {x} та {z}")

process_tuple((1, 2, 3))
# Виводить: Перший та третій елементи кортежу: 1 та 3
```

3. Кортеж із невизначеною кількістю елементів:

```
def process_tuple(tup):
    match tup:
        case (x, *rest):
            print(f"Перший елемент: {x}, решта елементів: {rest}")

process_tuple((1, 2, 3, 4, 5))
# Виводить: Перший елемент: 1, решта елементів: [2, 3, 4, 5]
```

16.4. Масиви в pattern matching

Декілька прикладів використання масивів у *pattern matching*:

1. Порівняння списку з конкретною структурою:

```
def process_list(lst):
    match lst:
        case [0, 1, 2]:
            print("Список містить елементи [0, 1, 2]")
        case [x, y, z]:
            print(f"Перший елемент: {x}, другий елемент: {y}, третій елемент: {z}")
        case _:
            print("Список має іншу структуру")

process_list([1, 2, 3])
# Виводить: Перший елемент: 1, другий елемент: 2, третій елемент: 3
```

2. Порівняння списку з використанням wildcard-ів та умов:

```
def process_list(lst):
    match lst:
        case [0, _, z] if z % 2 == 0:
            print("Список починається з 0, другий елемент будь-який, а третій парний")
        case [_, *_]:
            print("Список містить принаймні один елемент")
        case _:
            print("Список порожній")

process_list([0, 5, 4])
# Виводить: Список починається з 0, другий елемент будь-який, а третій парний
```

3. Порівняння списку з невизначеною кількістю елементів:

```
def process_list(lst):
    match lst:
        case [x, *rest]:
            print(f"Перший елемент: {x}, решта елементів: {rest}")

process_list([1, 2, 3, 4, 5])
# Виводить: Перший елемент: 1, решта елементів: [2, 3, 4, 5]
```

16.5. Масиви невизначеної довжини, альтернативні значення

У Python 3.10 конструкція *match* може бути використана для обробки масивів (списків) невизначеної довжини та використання альтернативних значень для шаблонів.

1. Масив невизначеної довжини:

```
def process_list(lst):
    match lst:
        case []:
            print("Список порожній")
        case [x, *rest]:
            print(f"Перший елемент: {x}, решта елементів: {rest}")

process_list([1, 2, 3, 4, 5])
# Виводить: Перший елемент: 1, решта елементів: [2, 3, 4, 5]
```

У цьому прикладі, шаблон [*x, *rest*] відповідає списку з першим елементом `*x*`, а решта елементів зберігається у змінній *rest*.

2. Альтернативні значення для шаблонів:

```
def process_list(lst):
    match lst:
        case [] | [0]:
            print("Список порожній або містить тільки 0")
        case [1, 2, 3] | [3, 2, 1]:
            print("Список містить послідовні числа 1, 2, 3 або 3, 2, 1")
        case _:
            print("Інша структура списку")

process_list([3, 2, 1])
# Виводить: Список містить послідовні числа 1, 2, 3 або 3, 2, 1
```

У цьому прикладі використано альтернативні значення для шаблонів за допомогою оператора `|`, що дає змогу вказати кілька можливих шаблонів для порівняння. Код виконується, якщо вхідний список відповідає будь-якому з цих шаблонів.

16.6. Словники в pattern matching

Приклади використання словників *у pattern matching*:

1. Порівняння словника з конкретними ключами та значеннями:

```
def process_dict(dct):
    match dct:
        case {'name': 'John', 'age': 30}:
            print("Словник містить ім'я 'John' та вік 30")
        case {'name': name, 'age': age}:
            print(f"Ім'я: {name}, Вік: {age}")
        case _:
            print("Інша структура словника")

process_dict({'name': 'Alice', 'age': 25})  # Виводить: Ім'я: Alice, Вік: 25
```

2. Порівняння словника з використанням *wildcard*[4] та умов:

```
def process_dict(dct):
    match dct:
        case {'name': _, 'age': age} if age >= 18:
            print(f"Словник містить особу з віком {age} років або старше")
        case _:
            print("Інша структура словника")

process_dict({'name': 'Bob', 'age': 20})  # Виводить: Словник містить особу з віком 20 років або старше
```

[4] У Python "wildcard" (символ заміщення) зазвичай використовується для опису спеціальних символів або конструкцій, які дають змогу узагальнювати пошук або відповідність. Wildcard символи в Python можна зустріти в різних контекстах.

3. Порівняння словника з невизначеною кількістю ключів:

```
def process_dict(dct):
    match dct:
        case {'name': name, rest}:
            print(f"Ім'я: {name}, решта ключів: {rest}")

process_dict({'name': 'Chris', 'age': 35, 'city': 'New York'})
# Виводить: Ім'я: Chris, решта ключів: {'age': 35, 'city': 'New York'}
```

4. Передача набору значень.

Конструкція *match* може бути використана для порівняння не тільки одного значення, але й набору значень. Це дає змогу легко обробляти різні комбінації значень, передаючи їх у конструкцію *match*.

```
def process_values(a, b):
    match (a, b):
        case (0, 0):
            print("Обидва значення дорівнюють 0")
        case (0, _):
            print("Перше значення дорівнює 0, друге значення будь-яке")
        case (_, 0):
            print("Перше значення будь-яке, друге значення дорівнює 0")
        case (_, _):
            print("Обидва значення будь-які")

process_values(0, 0)  # Виводить: Обидва значення дорівнюють 0
process_values(0, 5)  # Виводить: Перше значення дорівнює 0, друге значення будь-яке
process_values(3, 0)  # Виводить: Перше значення будь-яке, друге значення дорівнює 0
process_values(3, 5)  # Виводить: Обидва значення будь-які
```

У цьому прикладі конструкція *match* порівнюється з набором значень *(a, b)*, де *a* та *b* – це аргументи функції. Кожна клауза *case* визначає, які значення приймають *a* та *b*, і відповідний блок коду виконується, якщо значення відповідають шаблону.

16.7. Отримання значень ключів, отримання всіх значень

Конструкція *match* може бути використана для отримання значень ключів та отримання всіх значень словника. Це дає змогу легко отримати доступ до ключів та значень словника й обробляти їх у конструкції *match*.

1. Отримання значень ключів:

```
def process_dict(dct):
    match dct:
        case {'name': name, 'age': age}:
            print(f"Ім'я: {name}, Вік: {age}")
        case {'name': name}:
            print(f"Тільки ім'я: {name}")
        case {'age': age}:
            print(f"Тільки вік: {age}")
        case _:
            print("Інші ключі")

process_dict({'name': 'Alice', 'age': 25, 'city': 'New York'})
# Виводить: Ім'я: Alice, Вік: 25
```

У цьому прикладі конструкція *match* порівнюється зі словником *dct*. Кожна клауза `case` визначає, які ключі мають бути в словнику, та прив’язує їх значення до змінних, які можна використовувати у відповідному блоці коду.

2. Отримання всіх значень:

```
def process_dict(dct):
    match dct.items():
        case [('name', name), ('age', age)]:
            print(f"Ім'я: {name}, Вік: {age}")
        case _:
            print("Інші значення")

process_dict({'name': 'Alice', 'age': 25, 'city': 'New York'})
# Виводить: Ім'я: Alice, Вік: 25
```

У цьому прикладі конструкція *match* порівнюється зі списком пар ключ-значення, отриманим від методу *items()* словника. Кожна клауза *case* визначає, які пари ключ-значення мають бути в словнику, та прив’язує їх значення до змінних, які можна використовувати у відповідному блоці коду.

Перелік питань для самоконтролю

1. Конструкція *match*.
2. Кортежі в *pattern matching*.
3. Альтернативні значення, пропуск елементів, кортеж із невизначеною кількістю елементів.
4. Масиви в *pattern matching*.
5. Масиви невизначеної довжини, альтернативні значення.
6. Словники в *pattern matching*.
7. Передача набору значень.
8. Отримання значень ключів, отримання всіх значень.

Тестові питання

1. Що таке "pattern matching" у Python:

а) процес порівняння рядків з регулярними виразами;

б) нова концепція уведена в Python 3.10 для структурованого порівняння даних;

в) метод порівняння шаблонів у текстовому рядку;

г) властивість об′єктів класу Pattern у бібліотеці re.

2. Яка версія Python вперше включила підтримку pattern matching:

а) Python 3.9;

б) Python 3.8;

в) Python 3.10;

г) Python 3.7.

3. Як називається ключове слово, уведене в Python 3.10 для виконання pattern matching:

а) match;

б) pattern;

в) find;

г) select.

4. Які об’єкти можуть бути порівнюваними за допомогою pattern matching:

а) тільки рядки;

б) тільки списки;

в) будь-які об’єкти;

г) тільки числа.

5. Що таке "case" у виразі pattern matching:

а) пара значень, яка порівнюється з об’єктом;

б) оператор, який виконує порівняння за шаблоном;

в) блок коду, який виконується під час співпадіння з шаблоном;

г) усі зазначені вище.

6. Які з наведених операторів використовуються в pattern matching для визначення порівняння зі значенням:

а) ==;

б) :=;

в) =;

г) is.

7. Що повертається, якщо жоден шаблон не співпадає з виразом у виразі pattern matching:

а) SyntaxError;

б) TypeError;

в) ValueError;

г) None.

8. Який із наведених методів потрібно використовувати для визначення pattern matching у Python 3.10:

а) match();

б) find();

в) re();

г) case().

9. Яка з наведених функцій може бути використана з pattern matching для розгалуження коду:

а) if-else;

б) switch-case;

в) while;

г) for.

10. Яка структура даних часто використовується з pattern matching для розбору даних:

а) JSON;

б) Tuple;

в) Set;

г) Dictionary.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть функцію `number_type(n)`, яка використовує конструкцію `match` для визначення типу числа: додатнє, від'ємне або нуль.

2. Напишіть функцію `data_type(value)`, яка визначає тип даних (число, рядок, список, тощо) за допомогою `match`.

3. Створіть функцію `coordinate_description(coord)`, яка обробляє кортежі з координатами `(x, y)`.

4. Напишіть функцію `list_info(lst)`, яка перевіряє, чи список порожній або має певну кількість елементів.

5. Напишіть функцію `dict_description(d)`, яка перевіряє наявність певних ключів у словнику.

6. Напишіть функцію `extract_first_word(text)`, яка виділяє перше слово з рядка.

7. Створіть функцію `group_by_type(items)`, яка групує елементи списку за типами даних.

8. Напишіть функцію `is_prime_number(n)`, яка перевіряє, чи є число простим.

9. Створіть функцію `check_date_format(date_str)`, яка перевіряє, чи рядок є датою у форматі YYYY-MM-DD.

10. Напишіть функцію `process_input(data)`, яка обробляє числові й рядкові значення по-різному.

11. Напишіть функцію `parse_tuple(data)`, яка витягує значення з тривимірного кортежу `(x, y, z)`.

12. Напишіть функцію `even_or_odd(n)`, яка визначає, чи є число парним або непарним.

Тема 17
Виключення та обробка помилок

17.1. Помилки в програмі Python та методи обробки виключень.

17.2. Опис типів помилок у програмі Python. Інструкція try...except...else...finally. Формати інструкції try.

17.3. Поняття про протокол менеджерів контексту. Інструкція with...as. Формати інструкції with...as.

17.4. Конструкція with … open() та її використання під час обробки виключень.

17.5. Виключення користувача. Застосування інструкцій raises та assert для створення виключень користувача.

17.6. Основні поняття про клас Exception та його атрибути.

17.1. Помилки в програмі Python та методи обробки виключень

Помилки в програмі Python, також відомі як виключення, виникають у випадку неможливості виконання певного коду, наприклад, ділення на нуль або спроба доступу до неіснуючого елементу списку. Обробка виключень у Python дає змогу реагувати на ці помилки і виконувати відповідні дії для вирішення проблеми або її відстеження. Ось деякі методи обробки виключень у Python:

1. Блок *try-except*. Використовується для обробки виключень. Код, який може викликати помилку, поміщується в блок *try*, а обробка помилки відбувається в одному або декількох блоках *except*.

```
try:
    # Код, який може викликати помилку
    result = 10 / 0
except ZeroDivisionError:
    # Обробка помилки ділення на нуль
    print("Помилка: ділення на нуль")
```

2. Блок *try-except-finally*. Дає змогу виконати певні дії навіть у випадку виникнення виключення. Код, який викликає помилку, поміщується в блок *try*, обробка помилки – у блоках *except*, а код, що виконується завжди, – у блоці *finally*.

```
try:
    # Код, який може викликати помилку
    result = 10 / 0
except ZeroDivisionError:
    # Обробка помилки ділення на нуль
    print("Помилка: ділення на нуль")
finally:
    # Код, який виконується завжди
    print("Завершення обробки виключення")
```

3. Блок *try-except-else*. Дає змогу виконати певний код, якщо в блоці *try* не виникло виключення. Цей код поміщується в блок *else*.

```
try:
    # Код, який може викликати помилку
    result = 10 / 2
except ZeroDivisionError:
    # Обробка помилки ділення на нуль
    print("Помилка: ділення на нуль")
else:
    # Код, який виконується, якщо виключення не виникло
    print("Результат:", result)
```

4. Виклик виключень з власних функцій за допомогою *raise*: Розробники можуть викликати власні виключення для позначення специфічних помилок, які виникли в їх функціях.

```
def calculate_age(year):
    if year <= 0:
        raise ValueError("Рік повинен бути додатнім числом")
    return 2024 - year
```

```
try:
    age = calculate_age(1990)
    print("Вік:", age)
except ValueError as e:
    print("Помилка:", e)
```

Ці методи дають змогу ефективно управляти виключеннями в програмах Python і забезпечують гнучкість у вирішенні проблем, що можуть виникнути під час виконання програми.

17.2. Опис типів помилок у програмі Python. Інструкція try...except...else...finally. Формати інструкції try

У Python існує кілька типів помилок, які можуть виникати під час виконання програми.

Основні типи помилок у програмі Python:

1. *SyntaxError* (Помилка синтаксису). Виникає, коли в коді є синтаксична помилка, така як неправильно розміщені ключові слова, оператори або знаки пунктуації.

2. *IndentationError* (Помилка відступів). Виникає, коли рівень відступу в коді не відповідає очікуваному блоку коду, наприклад, у випадку неправильного використання пробілів або табуляцій.

3. *NameError* (Помилка імені). Виникає, коли використовується змінна або функція, яка не була визначена.

4. *TypeError* (Помилка типу). Виникає, коли операція виконується над об’єктом неправильного типу або коли функція отримує неправильну кількість аргументів або аргументи неправильного типу.

5. *ValueError* (Помилка значення). Виникає, коли функція отримує аргумент правильного типу, але з неприпустимим значенням.

Інструкція *try...except...else...finally* використовується для обробки виключень у Python.

```
try:
    # Блок коду, який може викликати помилку
except <тип_помилки> as <змінна_помилки>:
    # Блок коду, що обробляє виключення певного типу
else:
    # Блок коду, який виконується, якщо виключення не виникло
finally:
    # Блок коду, який завжди виконується
```

1. Спроба ділення на нуль:

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print("Помилка ділення на нуль:", e)
else:
    print("Результат:", result)
finally:
    print("Завершення обробки виключення")
```

2. Робота зі списком:

```
my_list = [1, 2, 3]
try:
    print(my_list[4])
except IndexError as e:
    print("Помилка індексу:", e)
else:
    print("Все пройшло успішно")
finally:
    print("Завершення обробки виключення")
```

3. Конвертація рядка в число:

```
try:
    number = int("abc")
except ValueError as e:
    print("Помилка конвертації:", e)
else:
    print("Число:", number)
finally:
    print("Завершення обробки виключення")
```

4. Виклик власного виключення:

```
def calculate_age(year):
    if year <= 0:
        raise ValueError("Рік повинен бути додатнім числом")
    return 2024 - year

try:
    age = calculate_age(1990)
    print("Вік:", age)
except ValueError as e:
    print("Помилка обчислення віку:", e)
```

5. Читання файлу:

```
try:
    with open("non_existent_file.txt", "r") as file:
        content = file.read()
except FileNotFoundError as e:
    print("Помилка читання файлу:", e)
else:
    print("Зміст файлу:", content)
finally:
    print("Завершення обробки виключення")
```

Ці приклади демонструють різні сценарії використання інструкції *`try...except...else...finally`* для обробки різних типів помилок у Python.

17.3. Поняття про протокол менеджерів контексту. Інструкція with...as. Формати інструкції with...as

Протокол менеджерів контексту в Python дає змогу автоматизувати ресурси або виконати певні дії під час входу в контекст і виходу з нього. Це дуже корисний механізм, особливо коли маємо справу з відкриттям та закриттям файлів, роботою з мережевими з'єднаннями або збиранням сміття.

Інструкція *with...as* використовується для створення контексту, в якому відбувається автоматичне керування ресурсами.

with <вираз_контексту> as <змінна>:

 # Блок коду, що виконується в контексті

У цьому форматі:

- *`<вираз_контексту>`* – це вираз, який повертає об'єкт менеджера контексту.

- *`<змінна>`* – це змінна, до якої прив'язується результат виклику *__enter__()* методу менеджера контексту.

- Блок коду виконується в контексті, керованому менеджером контексту.

Після виконання блоку коду метод *__exit__()* менеджера контексту викликається для звільнення ресурсів або виконання додаткових дій.

Отже, коли ви використовуєте *with...as*, ви гарантовано матимете правильне відкриття та закриття ресурсів.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# Файл автоматично закривається після виходу з контексту
```

У цьому прикладі *open("example.txt", "r")* повертає файловий об’єкт, який є менеджером контексту. Після виконання блоку коду файл автоматично закривається, навіть якщо виникає помилка.

Інструкція *with...as* дає змогу уникнути зайвого коду для відкриття та закриття файлів, а також забезпечує безпечне використання ресурсів у Python.

17.4. Конструкція with … open() та її використання під час обробки виключень

Конструкція *with ... open()* у Python використовується для відкриття файлів і автоматичного їх закриття після завершення відповідного блоку коду. Це забезпечує безпеку та ефективне використання файлових ресурсів, особливо коли маємо справу з обробкою виключень.

```
try:
    with open("example.txt", "r") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("Файл не знайдено")
except PermissionError:
    print("Відсутні права на доступ до файлу")
except Exception as e:
    print("Сталася невідома помилка:", e)
```

У цьому прикладі:

- ми відкриваємо файл "*example.txt*" у режимі читання за допомогою *with open("example.txt", "r") as file;*

- блок коду, який виконується всередині *with*, прочитує вміст файлу та виводить його на екран;

- якщо виникає виняток під час відкриття або читання файлу, програма переходить до відповідного блоку *except* для обробки помилки;

- у будь-якому випадку після виходу з блоку *with* файл автоматично закривається, навіть якщо виникає помилка.

Використання *with open()* спрощує код і гарантує правильне закриття файлу, навіть якщо сталася помилка в обробці відповідного блоку коду.

17.5. Виключення користувача. Застосування інструкцій raises та assert для створення виключень користувача

Виключення користувача в Python – це винятки, які створюються користувачем програми, коли вони потребують спеціальної поведінки або сигналізують про специфічні умови в програмі, які вимагають уваги.

Існують кілька способів створення власних виключень користувача в Python:

1. Створення власного класу виключення:

```
class CustomError(Exception):
    pass

# Використання власного виключення:
raise CustomError("Це моя власна помилка")
```

2. Наслідування від існуючих класів виключень:

```
class CustomError(ValueError):
    pass

# Використання власного виключення:
raise CustomError("Це моя власна помилка")
```

3. Підкласування інших виключень:

```
class CustomError(ZeroDivisionError):
    pass

# Використання власного виключення:
raise CustomError("Це моя власна помилка")
```

4. Створення екземпляра виключення з додатковими параметрами:

```
class CustomError(Exception):
    def __init__(self, message, code):
        super().__init__(message)
        self.code = code

# Використання власного виключення:
raise CustomError("Це моя власна помилка", 500)
```

Використання власних виключень користувача дає змогу створювати чітку інтерфейсну точку для обробки специфічних ситуацій або помилок у програмі. Під час обробки власних виключень потрібно враховувати потреби програми й забезпечити адекватну поведінку для користувачів або інших частин програми, які будуть обробляти ці виключення.

Інструкція *raise* використовується для явного виклику виключень у Python. Вона дає змогу створити та викликати виключення користувача у відповідь на певні умови або події.

Інструкція *assert* також може бути використана для створення виключень користувача, але зазвичай вона використовується для перевірки умов та викидання виключення, якщо умова не виконується.

1. Використання raise:

```
def calculate_area(length, width):
    if length <= 0 or width <= 0:
        raise ValueError("Довжина та ширина повинні бути
додатніми числами")
    return length * width

try:
    area = calculate_area(-5, 10)
    print("Площа:", area)
except ValueError as e:
    print("Помилка обчислення площі:", e)
```

У цьому прикладі функція *calculate_area* перевіряє, чи додатні значення передані їй як довжина та ширина. Якщо хоча б одне значення від'ємне або дорівнює нулю, вона викидає виключення типу *ValueError* з відповідним повідомленням про помилку.

2. Використання assert:

```
def calculate_area(length, width):
    assert length > 0 and width > 0, "Довжина та ширина повинні бути додатніми числами"
    return length * width

try:
    area = calculate_area(-5, 10)
    print("Площа:", area)
except AssertionError as e:
    print("Помилка обчислення площі:", e)
```

У цьому прикладі функція *calculate_area* використовує *assert*, щоб перевірити, чи додатні значення передані їй як довжина та ширина. Якщо умова не виконується, вона викидає виключення типу *AssertionError* з відповідним повідомленням про помилку.

Обидва ці підходи можуть бути використані для створення власних виключень користувача в Python залежно від потреб вашої програми та стилю коду.

17.6. Основні поняття про клас Exception та його атрибути

Клас *Exception* є базовим класом для всіх виключень у Python. Він є підкласом класу *BaseException* і є основою для створення власних типів виключень.

1. Ієрархія класів виключень. Клас *Exception* утворює основу ієрархії класів виключень у Python. Усі вбудовані та користувацькі виключення повинні бути підкласами класу *Exception*.

2. Власне виключення. Класи, які успадковані від *Exception*, можуть використовуватися для створення власних типів виключень, що відповідають певним умовам або помилкам у програмі.

3. Корисні атрибути:

- *args* – кортеж, що містить аргументи, передані в конструктор виключення;

- message – повідомлення про помилку, передане в конструктор виключення, у більшості випадків це є перший аргумент кортежа *args*;

- *__str__* – метод, який повертає рядкове представлення виключення. Зазвичай використовується для виведення повідомлення про помилку під час обробки виключення.

Створення власних виключень. Щоб створити власне виключення, потрібно створити новий клас, який успадкований від класу *Exception*. Можна також перевизначити методи класу *Exception*, такі як *__init__*, для додавання додаткових параметрів або логіки обробки виключення.

Отже, клас *Exception* є основою для обробки виключень у Python і надає зручний спосіб створення власних типів виключень з необхідними функціональними можливостями.

Метод *__exit__()* та його формат. Метод *__exit__()* є частиною протоколу менеджерів контексту в Python і використовується для виконання певних дій після виходу з контексту, незалежно від того, чи виникла помилка, чи ні. Цей метод викликається автоматично під час завершення роботи з контекстом, навіть якщо виникла помилка або була викликана інструкція *break*, *continue* або *return* у середині контексту.

Формат методу `*__exit__()*`наступний:

```
def __exit__(self, exc_type, exc_value, traceback):
    # Код для завершення роботи з контекстом
```

У цьому форматі:

- *exc_type* – тип виключення. Якщо жодне виключення не виникло, то *None*.

- *exc_value* – значення виключення, якщо воно виникло. Якщо жодне виключення не виникло, то *None*.

- *traceback* – об’єкт, який представляє відстеження стеку викликів програми. Цей аргумент використовується для виведення стеку викликів у випадку виникнення виключення.

Метод *__exit__()* може використовуватися для виконання різних дій, таких як закриття файлів, звільнення ресурсів або виведення додаткової інформації про помилку. Важливо пам’ятати, що метод *__exit__()* не повинен сам викидати виключення. Якщо ви хочете перехопити виключення в контексті і відобразити додаткову інформацію, це можна зробити у власному коді методу *__exit__()*, але виключення повинно бути викинуто знову після цього, якщо потрібно.

```
class CustomFileContext:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
        self.file.close()

# Використання контекстного менеджера для читання з файлу
with CustomFileContext("example.txt", "r") as file:
    content = file.read()
    print(content)
# Після виходу з контексту файл автоматично закривається
```

У цьому прикладі метод *__exit__()* викликає *close()* для закриття файлу після завершення роботи з контекстом.

Застосування функції *exc_info()* для одержання інформації про виключення.

Функція *exc_info()* з модуля *sys* у Python використовується для отримання інформації про поточне виключення, якщо таке виключення виникло у виконанні програми. Ця функція повертає кортеж, який містить три значення: тип виключення, об'єкт виключення і трейсбек.

Приклад використання функції *exc_info()* для отримання інформації про поточне виключення:

```
import sys

try:
    # Ділення на нуль, що призведе до виникнення виключення
    result = 10 / 0
except:
    exc_type, exc_obj, exc_tb = sys.exc_info()
    print("Тип виключення:", exc_type)
    print("Об'єкт виключення:", exc_obj)
    print("Стек викликів (traceback):", exc_tb)
```

У цьому прикладі, після того, як виникає помилка під час ділення на нуль, функція *exc_info()* використовується для отримання інформації про поточне виключення. Отримані значення зберігаються у змінних *exc_type*, *exc_obj* та *exc_tb*. Після цього можна використати ці значення для виведення інформації про виключення або для подальшої обробки в коді програми.

Зверніть увагу, що варто використовувати функцію *exc_info()* тільки в обробнику виключень, оскільки вона повертає інформацію про останній виняток, що був викликаний.

Перелік питань для самоконтролю

1. Помилки в програмі Python та методи обробки виключень.
2. Опис типів помилок у програмі Python.
3. Інструкція **try...except...else...finally**.
4. Формати інструкції **try**.

5. Поняття про протокол менеджерів контексту.

6. Інструкція **with...as.**

7. Формати інструкції with...as.

8. Конструкція **with open()** та її використання під час обробки виключень.

9. Виключення користувача.

10. Застосування інструкцій **raises** та **assert** для створення виключень користувача.

11. Основне поняття про клас **Exception** та його атрибути.

12. Метод **__exit__()** та його формат.

13. Застосування функції **exc_info()** для одержання інформації про виключення.

Тестові питання

1. Що таке виключення в Python:

а) спеціальні об’єкти, які позначають помилку в програмі;

б) спеціальні функції, які дають змогу обробляти виняткові ситуації;

в) сеціальні класи, які дають змогу створювати нові типи помилок;

г) спеціальні оператори, які дають змогу викидати виключення з програми.

2. Який оператор використовується для викидання виключення в Python:

а) try;

б) raise;

в) except;

г) throw.

3. Який блок коду використовується для обробки виключень у Python:

а) try;

б) catch;

в) except;

г) finally.

4. Які з наведених типів виключень є вбудованими в Python:

а) Warning;

б) Info;

в) Error;

г) Exception.

5. Як використовується блок коду `finally` у виключеннях у Python:

а) викликається, якщо виникає виключення;

б) викликається завжди, незалежно від того, чи сталася помилка;

в) викликається, якщо виняток не було оброблено;

г) викликається, якщо виняток було оброблено.

6. Який оператор використовується для створення власних типів виключень у Python:

а) try;

б) raise;

в) except;

г) assert.

7. Як визначити багаторазовий блок обробки виключень для різних типів помилок у Python:

а) try-except;

б) try-finally;

в) try-except-else;

г) try-except-finally.

8. Як отримати інформацію про виняток у Python:

а) звернутися до документації Python;

б) використати функцію `get_exception_info()`;

в) використати ключове слово `exception`;

г) використати функцію `sys.exc_info()`.

9. Який блок коду використовується для обробки виключень у Python, які виникають під час виконання коду:

а) try;

б) catch;

в) except;

г) finally.

10. Що робить ключове слово `assert` у Python:

а) перевіряє, чи виконується умова, і викидає виключення, якщо умова неправдива;

б) викидає виняток, якщо вказана умова істинна;

в) перевіряє, чи вказана умова істинна, і викидає виключення, якщо умова неправдива;

г) виконується завжди, незалежно від того, чи правдива умова.

Рекомендовані джерела:

Основні: 1; 2; 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть функцію `safe_divide(a, b)`, яка безпечно виконує ділення `a` на `b`, повертаючи результат або повідомлення про помилку, якщо `b` дорівнює нулю.

2. Напишіть функцію `convert_to_int(value)`, яка перетворює рядок у ціле число, обробляючи помилку, якщо перетворення не вдається.

3. Створіть функцію `read_file(filename)`, яка намагається прочитати файл і повертає його вміст або повідомлення про помилку, якщо файл не існує.

4. Напишіть функцію `get_value(d, key)`, яка безпечно отримує значення зі словника за ключем, повертаючи повідомлення про помилку, якщо ключ відсутній.

5. Напишіть функцію `calculate_square_root(x)`, яка обчислює квадратний корінь числа, обробляючи помилки для від'ємних чисел.

6. Напишіть декоратор `handle_exceptions`, який обробляє будь-яке виключення, що виникає під час виконання функції.

7. Напишіть функцію `validate_email(email)`, яка перевіряє формат електронної пошти, обробляючи помилки за неправильного формату.

8. Напишіть функцію `parse_json(json_str)`, яка намагається конвертувати рядок в об'єкт JSON, обробляючи помилки за некоректного JSON.

9. Напишіть функцію `get_element(lst, index)`, яка намагається отримати елемент за індексом у списку, обробляючи помилку, якщо індекс виходить за межі.

10. Напишіть функцію `write_to_file(filename, content)`, яка намагається записати вміст у файл, обробляючи помилки, якщо файл не може бути відкритий для запису.

11. Напишіть функцію `perform_operations(a, b)`, яка виконує кілька операцій, обробляючи помилки ділення на нуль і помилки перетворення типів.

12. Створіть функцію `access_resource(url)`, яка намагається підключитися до ресурсу й обробляє помилки, якщо ресурс недоступний або виникає інша помилка.

13. Напишіть функцію `connect_to_db(db_url)`, яка намагається підключитися до бази даних та обробляє помилки під час підключення.

Тема 18
Модулі та пакети в Python

18.1. Модулі.

18.2. Пакети.

18.3. Функція getattr( ).

18.4. Перевірка існування атрибута за допомогою функції hasattr( ).

18.5. Способи одержання скомпільованого файлу з розширенням *.pyc.

18.6. Шляхи пошуку модулів.

18.7. Повторне завантаження модулів.

18.1. Модулі

У Python модуль – це файл з розширенням «*.py*», який містить Python код. Цей файл може містити функції, класи, змінні та інші об’єкти Python, які можуть бути використані в інших програмах або модулях.

Основні характеристики модулів у Python.

1. Використання коду. Модулі дають змогу організовувати код у логічні групи. Це сприяє підтримці структурованого та організованого програмного коду.

2. Імпорт і використання. Імпорт модулів дає змогу використовувати їх функції, класи та змінні в інших програмах або модулях. Це сприяє повторному використанню коду та полегшує роботу з великими проєктами.

3. Ізоляція коду. Кожен модуль у Python ізольований, що дає змогу уникнути конфліктів імен та забезпечити чистоту коду.

4. Розширюваність. Модулі можуть бути легко розширені або змінені, що робить їх потужним інструментом для розвитку програм.

5. Пакети. Модулі можуть бути організовані в пакети, що дає змогу групувати їх та структурувати код проєкту більш ефективно.

Наприклад, якщо у вас є файл з назвою *module.py*, в якому міститься такий код:

```
# module.py

def greet(name):
    print("Hello, " + name)

def square(x):
    return x  2

pi = 3.14159
```

Ви можете імпортувати та використовувати цей модуль в інших програмах або модулях, наприклад:

```
# main.py

import module

module.greet("Alice")  # Виведе: Hello, Alice

result = module.square(5)
print(result)  # Виведе: 25

print(module.pi)  # Виведе: 3.14159
```

Отже, модулі дають змогу організовувати та використовувати Python код у ваших програмах ефективним та легким способом.

1. Головний модуль *__main__*.

У Python кожен файл модуля має змінну *__name__*, яка містить ім’я модуля. Однак існує спеціальний випадок, коли файл модуля використовується як головний файл програми – це коли він виконується безпосередньо, а не імпортується в інший файл. У такому випадку значення *__name__* для цього файлу буде `мати значення *__main__*. Це дає змогу виконувати додаткові дії, коли

файл модуля використовується як головний файл програми, наприклад, запуск функцій чи коду, які не повинні виконуватися при імпорті модуля.

Приклад, який ілюструє використання головного модуля *__main__*:

```
# Файл module.py

def greet():
    print("Hello from module!")

if __name__ == "__main__":
    # Викликаємо функцію greet(),
# якщо цей файл модуля використовується як головний файл програми
    greet()
```

Тепер, якщо запустити файл *module.py* напряму, то ви побачите повідомлення *"Hello from module!"*. Однак, якщо ви імпортуєте *module.py* в інший файл, функція *greet()* не буде викликана.

```
# Файл main.py
import module

# Функція greet() не буде викликана, тому що module.py імпортується,
# а не виконується напряму
```

Використання *__name__ == "__main__"* дає змогу створювати модулі, які можуть використовуватися як самостійні програми або як частини більших проєктів, забезпечуючи гнучкість та перевірку безпеки виконання коду.

2. Атрибут об’єкта модуля *__main__: __name__*.

Атрибут об’єкта модуля *__main__: __name__* є одним з основних атрибутів у Python, який має тип *str* і використовується для ідентифікації модуля.

Коли Python інтерпретує файл модуля, він присвоює атрибуту *__name__* значення *"__main__"*. Це стає можливим завдяки тому, що Python розпізнає, що саме цей файл виконується як головний програмний файл. Це дає змогу виконувати певні дії залежно від того, чи файл модуля використовується як головний файл програми, або ж як модуль, що імпортується в інші файли.

Потрібно використовувати конструкцію *if __name__ == "__main__":*, щоб викликати певні функції або виконати певні операції, коли файл модуля використовується як головний файл програми.

3. Інструкція *import* та варіанти її застосування для імпортування модулів.

Інструкція *import* використовується в Python для імпортування модулів, щоб використовувати їх функції, класи та інші об'єкти в поточному файлі або програмі. Існують різні варіанти застосування інструкції *import*, які дають змогу контролювати, як саме модуль імпортується та які об'єкти доступні після імпортування.

1. Імпорт модуля цілком:

```
import module
```

Цей варіант імпортує весь модуль цілком, і всі його об'єкти доступні після префікса імені модуля.

2. Імпорт конкретного об'єкта з модуля:

```
from module import object_name
```

Цей варіант імпортує лише конкретний об'єкт (функцію, клас або змінну) з модуля, щоб він був доступний без префікса імені модуля.

3. Імпорт кількох об'єктів з модуля:

```
from module import object1, object2, ...
```

Цей варіант дає змогу імпортувати кілька об'єктів з модуля одночасно.

4. Імпорт модуля з альтернативним ім’ям:

```
import module as alt_name
```

Цей варіант імпортує модуль, але присвоює йому альтернативне ім’я, щоб використовувати його під іншою назвою.

5. Імпорт усіх об’єктів модуля:

```
from module import *
```

Цей варіант імпортує всі об’єкти з модуля, але цей спосіб не рекомендується через можливість конфліктів імен.

6. Імпорт модуля з використанням релятивного шляху:

```
from . import module
```

Цей варіант використовується для імпортування модуля з поточного пакета.

Інструкція *import* дає змогу зручно керувати тим, як саме модуль і його об’єкти будуть імпортовані та використані у вашій програмі.

18.2. Пакети

У Python пакет є простором імен, який містить колекцію модулів та підпакетів. Це організаційний метод, що дає змогу структурувати та організовувати великі програмні проєкти, розподіляючи їх на логічні групи модулів.

Основні характеристики пакетів у Python:

1. Каталог з Python файлами. Пакет зазвичай є каталогом, який містить файл __init__.py (ініціалізаційний файл пакета) та інші Python файли, які складаються з модулів.

2. Простір імен. Кожен пакет створює свій власний простір імен, що дає змогу уникнути конфліктів імен між модулями в різних пакетах.

3. Ієрархічна організація. Пакети можуть містити підпакети, що дає змогу створювати ієрархічну організацію проєкту.

4. Ініціалізаційний файл __init__.py. Файл __init__.py може містити код, який виконується під час імпортування пакета, і він також вказує Python, що каталог є пакетом. Цей файл може бути порожнім або містити ініціалізаційний код.

Наприклад, пакет з назвою `my_package`, який містить два модулі `module1.py` та `module2.py`:

```
my_package/
│
├── __init__.py
├── module1.py
└── module2.py
```

У файлі *__init__.py* може бути, наприклад, такий вміст:

```
# __init__.py

print("Initializing my_package")

# Можливо, додатковий код ініціалізації пакета
```

Тепер можна імпортувати цей пакет та його модулі в іншому коді Python:

```
import my_package.module1
import my_package.module2

my_package.module1.some_function()
my_package.module2.some_other_function()
```

Використання пакетів дає змогу організувати та структурувати великі програмні проєкти в Python, щоб зробити їх більш зрозумілими та керованими.

Файл *__init__.py* в пакеті Python може містити будь-який код, який ви бажаєте виконати під час імпортування пакета.

1. Порожній файл ініціалізації.

2. Імпорт модулів відразу під час імпортування пакета:

```
# Файл __init__.py

from .module1 import *
from .module2 import some_function
```

3. Визначення змінних чи констант пакета:

```
# Файл __init__.py

PI = 3.14159
VERSION = "1.0"
```

4. Виконання додаткового коду під час імпортування пакета:

```
# Файл __init__.py

print("Initializing my_package")
```

5. Використання *__all__* для визначення списку модулів, доступних під час імпортування пакета:

```
# Файл __init__.py

__all__ = ["module1", "module2"]
```

Файл *__init__.py* може містити будь-яку комбінацію цих прикладів або іншого коду, який ви бажаєте виконати під час імпортування пакета. Важливою річчю є те, що наявність файлу *__init__.py* вказує Python, що каталог є пакетом, і цей файл буде виконаний під час кожного імпортування пакета.

5. Одержання доступу до ідентифікаторів імпортованого модуля.

Під час імпортування модуля в Python ви отримуєте доступ до його ідентифікаторів, таких як функції, змінні та класи. Існує кілька способів отримання доступу до цих ідентифікаторів:

1. Використання імені модуля як префікса:

```
import module

module.some_function()
module.some_variable
```

2. Використання оператора *from ... import*:

```
from module import some_function, some_variable

some_function()
print(some_variable)
```

3. Використання псевдонімів для імпорту:

```
import module as mod

mod.some_function()
```

4. Використання `*` для імпорту всіх ідентифікаторів модуля:

```
from module import *

some_function()
print(some_variable)
```

5. Доступ до підпакетів і модулів через простори імен:

```
import package.module

package.module.some_function()
```

6. Імпорт і використання ідентифікаторів модуля в одному рядку:

```
from module import some_function as func, some_variable as var

func()
print(var)
```

Ці методи дають змогу ефективно використовувати функції, змінні та класи з імпортованого модуля у програмі. Вибір конкретного методу залежить від вашого стилю програмування та потреб проєкту.

6. Імпорт з головного модуля та з файлів пакета.

Імпорт з головного модуля й файлів пакета виконується подібно, але є різниця у шляхах, які використовуються для імпортування.

1. Імпорт з головного модуля:

Якщо у вас є файл, який виконується напряму (головний модуль), ви можете імпортувати модулі та об'єкти з інших модулів, включаючи файли з пакетів, за допомогою звичайних інструкцій *import* або *from ... import*.

Нехай є файл з назвою *main.py*, який є головним модулем, і він містить такий код:

```
# main.py

import module  # Імпорт з модуля

from package import sub_module  # Імпорт з підпакета

module.some_function()  # Виклик функції з модуля
sub_module.another_function()  # Виклик функції з підпакета
```

2. Імпорт з файлів пакета:

Під час імпортування з файлів пакета потрібно вказати шлях до файлу в межах пакета. Це може бути зроблено через крапки або за допомогою ключових слів *from ... import* для відносного або абсолютного імпортування відповідно.

Припустимо, що є пакет з назвою *package*, який містить файли *sub_module.py* та *sub_module2.py*. Можна імпортувати їх у головному модулі або в інших модулях пакета так:

```
# Відносний імпорт з одного файлу пакета в іншому файлі пакета
from . import sub_module

# Відносний імпорт з одного файлу пакета в іншому файлі пакета
from .sub_module import some_function

# Абсолютний імпорт з іншого файлу пакета
from package.sub_module2 import another_function
```

18.3. Функція getattr( )

Функція *getattr()* для динамічного формування назви атрибута в ході виконання програми.

Комбінування модулів, пакетів та функції *getattr()* дає змогу створювати дуже гнучкі та універсальні рішення для роботи з програмами. Припустимо, що є пакет *my_package*, що містить модуль *my_module.py*, а також файл *config.py*, де зберігаються різні конфігураційні параметри.

```
my_package/
│
├── __init__.py
├── my_module.py
└── config.py
```

У файлі *config.py* ми можемо мати різні конфігураційні параметри, представлені як атрибути модуля:

```
# config.py

DATABASE_HOST = "localhost"
DATABASE_PORT = 5432
DEBUG_MODE = False
```

Тепер у файлі *my_module.py* ми можемо використати функцію *getattr()* для динамічного отримання цих атрибутів під час виконання програми:

```
# my_module.py

import config

def get_config_value(attribute_name):
    # Використання getattr() для отримання значення атрибута
    value = getattr(config, attribute_name, None)
    return value

# Приклад використання
host = get_config_value('DATABASE_HOST')
port = get_config_value('DATABASE_PORT')
debug = get_config_value('DEBUG_MODE')

print("Database Host:", host)
print("Database Port:", port)
print("Debug Mode:", debug)
```

Цей підхід дає змогу легко змінювати конфігураційні параметри в файлі *config.py*, не змінюючи коду в інших частинах програми. Крім того, можна динамічно вибирати атрибути для отримання під час виконання програми, що робить код більш гнучким та універсальним.

18.4. Перевірка існування атрибута за допомогою функції hasattr( )

Функція *hasattr()* у Python дає змогу перевіряти, чи існує атрибут з певною назвою в об’єкті. Це корисно, коли потрібно перевірити наявність певного атрибута перед його використанням, особливо якщо ви працюєте з об’єктами, які мають динамічно визначені атрибути або які можуть не мати всіх атрибутів.

hasattr(object, attribute_name)

де:

- *object* – об’єкт, для якого ви хочете перевірити наявність атрибута;

- *attribute_name* – рядок, що містить ім’я атрибута, який ви хочете перевірити.

Функція *hasattr()* повертає *True*, якщо атрибут існує у вказаному об’єкті, і *False*, якщо атрибут не існує.

Ось приклад використання *hasattr()* для перевірки наявності атрибута в пакеті та модулі:

```
# Перевірка існування атрибута в пакеті
import my_package

if hasattr(my_package, 'some_attribute'):
    print("Attribute 'some_attribute' exists in my_package")
else:
    print("Attribute 'some_attribute' does not exist in my_package")

# Перевірка існування атрибута в модулі
import my_module

if hasattr(my_module, 'some_function'):
    print("Attribute 'some_function' exists in my_module")
else:
    print("Attribute 'some_function' does not exist in my_module")
```

Цей код перевіряє наявність атрибутів у пакеті *my_package* та модулі *my_module*. Якщо атрибут існує, виводиться повідомлення про його існування, інакше виводиться повідомлення про його відсутність.

18.5. Способи одержання скомпільованого файлу з розширенням *.pyc

У Python файл з розширенням *.pyc* є скомпільованою версією вихідного коду Python. Цей файл зазвичай створюється автоматично під час імпортування модуля або під час виконання скрипту Python. Ось кілька способів, які можна використовувати для отримання скомпільованого файлу *.pyc*:

1. Імпортування модуля:

Коли ви імпортуєте модуль у Python, інтерпретатор автоматично шукає відповідний файл *.pyc*. Якщо він знаходиться, він використовує його. Якщо файл *.pyc* відсутній або застарілий, Python створить новий файл *.pyc* під час імпортування модуля.

2. Використання командного рядка Python:

Можна скомпілювати файл *.py* вручну, використовуючи командний рядок Python з флагом *-m* (який виконує модуль як скрипт) та модуль *compileall*, який скомпілює всі файли Python у поточному каталозі:

```
python -m compileall your_directory
```

3. Використання модуля *py_compile*:

Модуль *py_compile* в Python надає функції для компіляції файлів Python у файл *.pyc*. Можна використати його в коді, щоб скомпілювати файл.

```
import py_compile

py_compile.compile('your_file.py')
```

4. Використання командного рядка Python з опцією *-m py_compile*:

Також можна використати командний рядок Python з модулем *py_compile*:

```
python -m py_compile your_file.py
```

18.6. Шляхи пошуку модулів

Шляхи пошуку модулів визначають, де інтерпретатор Python шукає модулі під час імпортування. Існують кілька стандартних місць, де інтерпретатор шукає модулі, а також можливості для користувача налаштовувати цей пошук.

1. Поточний каталог:

Інтерпретатор спочатку шукає модулі в поточному каталозі, де міститься запущений скрипт.

2. Змінна середовища *PYTHONPATH*:

Змінна середовища *PYTHONPATH* містить список каталогів, які Python переглядає для пошуку модулів. Якщо ця змінна визначена, інтерпретатор шукає модулі в каталогах, перерахованих у *PYTHONPATH*.

3. Стандартні каталоги інсталяції Python:

Python має свої власні стандартні каталоги, де розташовані вбудовані модулі та модулі стандартної бібліотеки. Інтерпретатор автоматично переглядає ці каталоги для пошуку модулів.

4. Сайти-пакети:

Це каталоги, які містять встановлені пакети Python. Вони можуть бути вказані в файлі конфігурації Python або встановлені системним менеджером пакетів. Python шукає модулі в цих каталогах після стандартних каталогів і перед каталогами, перерахованими у змінній середовища *PYTHONPATH*.

5. Модуль *site-packages*:

Це спеціальний каталог, де зазвичай встановлюються модулі сторонніх розробників або користувацькі модулі. Він знаходиться всередині каталогу з установленим Python та містить модулі, доступні для всіх користувачів.

Ці шляхи пошуку дають змогу інтерпретатору знаходити модулі, які потрібні для вашої програми. Якщо ви хочете додати власні шляхи пошуку або змінити порядок пошуку, ви можете скористатися змінною середовища *PYTHONPATH* або використати модуль *sys* для програмної конфігурації шляхів пошуку у вашому коді.

18.7. Повторне завантаження модулів

Повторне завантаження модулів у Python може бути корисним у випадках, коли потрібно оновити вміст модуля без перезапуску програми. Це може стати в нагоді під час розробки або в ситуаціях, коли зміни в модулі відбуваються під час виконання програми.

1. Використання функції *reload()* з модуля *imp*:

Модуль *imp* містить функцію *reload()*, яка дає змогу повторно завантажувати модуль. Однак потрібно звернути увагу, що модуль *imp* є застарілим з Python 3.4, і замість нього рекомендовано використовувати модуль *importlib.*

```
import imp

# Завантаження модуля
import my_module

# Повторне завантаження модуля
imp.reload(my_module)
```

2. Використання функції *reload()* з модуля *importlib*:

З модулем *importlib* у Python 3.4 і новіше ви можете також використовувати функцію *reload()*. Цей спосіб є більш сучасним і рекомендованим порівняно з *imp.reload()*.

```
import importlib

# Завантаження модуля
import my_module

# Повторне завантаження модуля
importlib.reload(my_module)
```

Ці два підходи дають змогу повторно завантажити модуль у програмі. Обирати потрібно той, який підходить до версії Python та відповідає вашим потребам. ***Увага! Повторне завантаження модуля може призвести до непередбачуваних результатів, якщо він використовується у великій програмі з багатою взаємодією між модулями***.

Зверніть увагу, що функція reload() повертає змінений об'єкт модуля, а не створює новий.

Перелік питань для самоконтролю

1. Визначення модуля в Python.
2. Головний модуль **"__main__".**
3. Атрибут об'єкта модуля **__main__: __name__.**
4. Інструкція **import** та варіанти її застосування для імпортування модулів.
5. Означення пакета.
6. Файл ініціалізації пакета **__init__.py**.
7. Одержання доступу до ідентифікаторів імпортованого модуля.
8. Імпорт з головного модуля та з файлів пакета.
9. Імпорт кількох модулів однією інструкцією.
10. Функція **getattr( )** для динамічного формування назви атрибута в ході виконання програми.
11. Перевірка існування атрибута за допомогою функції **hasattr( )**.
12. Способи одержання скомпільованого файлу з розширенням ***.pyc**.
13. Шляхи пошуку модулів.
14. Повторне завантаження модулів.
15. Формат функції **reload( )**.

Тестові питання

Звідки взяти модулі і пакети? Так, з Python, звісно! Ось декілька тестових питань, щоб перевірити знання про модулі та пакети в Python:

1. Що таке модуль у Python:

а) функція;

б) змінна;

в) файл з Python кодом;

г) клас.

2. Які з наведених ключових слів використовуються для імпорту модуля в Python:

а) `import`;

б) `use`;

в) `load`;

г) `get`.

3. Яка інструкція використовується для імпорту конкретної функції або змінної з модуля в Python:

а) `import`;

б) `include`;

в) `from`;

г) `require`.

4. Що таке пакет у Python:

а) функція;

б) змінна;

в) каталог, який містить модулі та інші пакети;

г) клас.

5. Яке розширення файлів зазвичай мають модулі Python:

а) `.py`;

б) `.pt`;

в) `.pm`;

г) `.pyt`.

6. Які імпортні операції потрібно використовувати, щоб імпортувати всі імена з модуля в Python:

а) `all`;

б) `import *`;

в) `all_names`;

г) `import all`.

7. Як виконати імпорт модуля під іншим ім'ям у Python:

а) `import module as new_name`;

б) `import as module new_name`;

в) `module as new_name import`;

г) `new_name import module`.

8. Що таке атрибут модуля в Python:

а) функція або змінна, які містяться всередині модуля;

б) змінна, яка міститься поза модулем;

в) функція, яка міститься поза модулем;

г) функція або змінна, які містяться поза модулем.

9. Яка функція в Python використовується для переліку всіх доступних атрибутів модуля:

а) `list_attributes()`;

б) `list(module)`;

в) `dir(module)`;

г) `module.attributes()`.

10. Яка різниця між модулем та пакетом у Python:

а) модуль – це файл з Python кодом, пакет – це каталог, що містить модулі та інші пакети;

б) модуль – це каталог, що містить функції та змінні, пакет – це файл з Python кодом;

в) модуль – це каталог, що містить модулі та інші пакети, пакет – це файл з Python кодом;

г) немає різниці, обидва терміни використовуються інтерченжевельно.

Рекомендовані джерела:
Основні: 1; 2; 3; 4.
Допоміжні: 1; 2; 3.
Інформаційні ресурси інтернет: 1; 2.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Створіть файл `math_utils.py`, який містить функцію `add(x, y)`, що повертає суму двох чисел. Використовуйте цей модуль у іншому скрипті, щоб підсумувати два числа.

2. Створіть файл `string_utils.py` з функцією `capitalize_words(text)`, яка перетворює першу літеру кожного слова на велику. Імплементуйте її в іншому скрипті.

3. Створіть файл `math_operations.py`, який містить функцію `subtract(a, b)`. Імплементуйте функцію в іншому скрипті, використовуючи псевдонім для модуля.

4. Створіть файл `utils.py` з кількома функціями: `square(x)`, `cube(x)`. Імплементуйте всі функції з модуля в іншому скрипті.

5. Додайте в `example.py` блок `if __name__ == "__main__":`, що виконує певний код за прямого запуску файлу. Перевірте поведінку під час імпорту модуля в іншому скрипті.

6. Створіть структуру пакета з двома модулями `module1.py` та `module2.py`, які містять функції `func1()` і `func2()` відповідно. Імплементуйте використання цих модулів у скрипті `main.py`.

7. Додайте файл `__init__.py` до пакета й використовуйте відносні імпорти для доступу до функцій у модулях.

8. Створіть підпакет `subpackage` з модулем `submodule.py`, що містить функцію `sub_func()`. Імплементуйте її у скрипті, використовуючи імпорт з підпакета.

9. Створіть модуль `error_demo.py` з помилковим імпортом та обробляйте можливі помилки під час імпорту.

10. Налаштуйте Python для імпорту модулів з нестандартних директорій за допомогою `sys.path`.

11. Напишіть модуль `shapes.py` з класом `Rectangle`, що має методи для розрахунку площі та периметра. Імплементуйте цей клас у скрипті.

12. Створіть модуль `generators.py` з генератором `count_up_to(n)` та використовуйте його у скрипті.

13. Створіть модуль `config.py` з налаштуваннями для програми та імплементуйте ці налаштування у скрипті.

14. Створіть модуль `exception_demo.py` з функцією, яка може кинути виключення, й обробляйте його у скрипті.

15. Створіть модуль `decorators.py` з декоратором, що логує виклики функцій, і застосуйте цей декоратор до функції у скрипті.

ЗМІСТОВИЙ МОДУЛЬ 2
ОСНОВИ ТА ПРИНЦИПИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ НА ПРИКЛАДІ МОВИ PYTHON

Тема 19
Поняття про об'єктно-орієнтоване програмування

19.1. Основні означення, що пов'язані з об'єктно-орієнтованим програмуванням.

19.2. Визначення класу й створення екземпляра класу.

19.3. Синтаксис створення класу та атрибута класу.

19.4. Спосіб створення методу класу за допомогою інструкції def.

19.5. Застосування змінної self для доступу до атрибутів та методів усередині класу.

19.6. Методи __init__() для задавання початкових значень атрибутам екземпляра класу.

19.7. Метод __del__() як деструктор, що викликається під час видалення екземпляра класу.

19.1. Основні означення, що пов'язані з об'єктно-орієнтованим програмуванням

Об'єктно-орієнтоване програмування (ООП). ООП – це парадигма програмування, яка використовує об'єкти та класи для організації коду. Основна ідея полягає в тому, щоб моделювати реальні об'єкти та їх взаємодію у вигляді об'єктів та класів у програмі.

2. Класи та об'єкти. Клас – це шаблон або заготовка для створення об'єктів. Він визначає атрибути та методи, які мають об'єкти класу. Об'єкт – це конкретний екземпляр класу, який має свої власні унікальні значення атрибутів та може викликати методи, визначені в класі.

3. Атрибути та методи. Атрибути – це дані, які належать об'єкту. Вони представлені як змінні, які зв'язані з конкретним об'єктом. Методи – це функції, які визначають поведінку об'єкта або виконують операції з його атрибутами.

4. Спадкування. Спадкування – це механізм, що дає змогу створювати нові класи на основі вже існуючих (базових) класів. Новий клас успадковує атрибути та методи базового класу і може розширювати або перевизначати їх.

5. Інкапсуляція. Інкапсуляція – це концепція, що дає змогу обмежувати доступ до деяких компонентів об'єкта та приховувати їх внутрішню реалізацію від зовнішнього світу. Це забезпечує безпеку та чистоту коду.

6. Поліморфізм. Поліморфізм – це можливість об'єктів різних класів використовувати однакові інтерфейси. Це дає змогу викликати один і той самий метод на об'єктах різних класів та мати різну реалізацію для кожного класу.

7. Абстракція. Абстракція – це процес визначення спільного інтерфейсу для групи схожих об'єктів або процесів. Вона дає змогу уникати деталей реалізації та концентруватися лише на важливих аспектах.

Ці поняття є основними та відображають основні концепції ООП, які використовуються в розробці програмного забезпечення з використанням Python.

19.2. Визначення класу й створення екземпляра класу

Клас – це шаблон або заготовка для створення об'єктів. Він визначає атрибути та методи, які мають об'єкти цього класу. Екземпляр класу – це конкретний об'єкт, створений на основі цього класу.

1. Побудова класу *Person* з атрибутами *name* та *age*:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Створення екземпляра класу Person
person1 = Person("John", 30)
print(person1.name)  # Виведе: John
print(person1.age)   # Виведе: 30
```

2. Побудова класу *Circle* з атрибутом *radius* та методом *area* для обчислення площі кола:

```
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius  2

# Створення екземпляра класу Circle
circle1 = Circle(5)
print(circle1.area())  # Виведе: 78.53981633974483
```

3. Побудова класу *Car* з атрибутами *make, model* та методом *display_info* для виведення інформації про автомобіль:

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def display_info(self):
        print(f"Car: {self.make} {self.model}")

# Створення екземпляра класу Car
car1 = Car("Toyota", "Camry")
car1.display_info()  # Виведе: Car: Toyota Camry
```

У цих прикладах ми визначаємо класи *Person, `Circle` та Car*, кожен з яких має свої атрибути та методи. Потім ми створюємо екземпляри цих класів (`person1`, `circle1`, `car1`) і використовуємо їх для доступу до атрибутів та виклику методів цих класів.

19.3. Синтаксис створення класу та атрибута класу

Створення класів у Python підлягає певним правилам і вимогам.

Основи створення класів:

1. Ключове слово *class*. Для оголошення класу використовують ключове слово *class*.

2. Назва класу. Імена класів зазвичай використовуються у стилі *CamelCase* (перше слово з великої літери, інші – з малих, без підкреслень), наприклад, *MyClass*.

3. Тіло класу. Містить методи та атрибути класу. Тіло класу визначається з відступом.

Синтаксис визначення класу:

```
class ClassName:
    def __init__(self, attribute):
        self.attribute = attribute

    def method(self):
        # Метод класу
        pass
```

Розгорнутий приклад:

```
class Person:
    def __init__(self, name, age):
        self.name = name  # Атрибут екземпляра
        self.age = age

    def greet(self):
        print(f"Привіт, мене звати {self.name} мені {self.age} років.")

    @staticmethod
    def species():
        return "Homo sapiens"
```

Основні компоненти класу:

1. Метод *__init__* (конструктор):

- ініціалізує атрибути об’єкта під час створення;
- має перший параметр *self*, що вказує на екземпляр класу.

2. Атрибути екземпляра. Змінні, що належать конкретному екземпляру класу, визначаються через *self*.

3. Методи класу:

- функції, що визначені всередині класу і виконуються в контексті екземпляра класу;
- перший параметр *self* посилається на сам об’єкт.

4. Методи класу та статичні методи:

- методи класу: визначаються за допомогою декоратора *@classmethod* і приймають перший параметр *cls*, що посилається на сам клас;
- статичні методи: визначаються за допомогою декоратора *@staticmethod* і не приймають жодних спеціальних параметрів.

5. Метод *__str__* і *__repr__*:

- *__str__*: Використовується для створення зручного для читання рядка, який представляє об’єкт;
- *__repr__*: Використовується для створення точного рядка, який може бути використаний для відновлення об’єкта.

6. Метод *__del__* (деструктор). Викликається, коли об’єкт видаляється. Зазвичай використовується рідко.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound.")

    @classmethod
    def general_info(cls):
        print("Animals are multicellular organisms.")
```

```
    @staticmethod
    def is_animal():
        return True

    def __str__(self):
        return f"Animal named {self.name}"

    def __repr__(self):
        return f"Animal(name={self.name!r})"
```

У Python атрибути класу визначаються всередині класу і призначаються об’єктам цього класу. Ось синтаксис створення атрибута класу:

```
class ClassName:
    attribute_name = value
```

де:

- *class* – ключове слово, що вказує на початок визначення класу;
- *ClassName* – ім’я класу, до якого належатиме атрибут;
- *attribute_name* – ім’я атрибута, яке ви визначаєте для класу;
- *value* – значення, яке ви призначаєте атрибуту. Це може бути будь-яке допустиме значення Python, таке як ціле число, рядок, список, кортеж, інший об’єкт тощо.

Наприклад:

```
class Person:
    name = "John"
```

Зверніть увагу, що атрибути класу є спільними для всіх екземплярів класу. Якщо ви зміните значення атрибута для одного об’єкта, це також вплине на інші об’єкти того самого класу.

19.4. Спосіб створення методу класу за допомогою інструкції def

Щоб створити метод класу за допомогою інструкції *def* у Python, можна просто визначити функцію всередині класу. Загальний синтаксис для цього:

```
class ClassName:
    def method_name(self, parameter1, parameter2, ...):
        # Тіло методу
        # Виконується код методу
```

- *class* – ключове слово, що вказує на початок визначення класу;
- *ClassName* – ім'я класу, до якого належатиме метод;
- *method_name* – ім'я методу, яке ви визначаєте для класу;
- *self* – обов'язковий параметр, який представляє поточний об'єкт класу. Він дає змогу методу отримати доступ до атрибутів та інших методів цього об'єкта;
- *parameter1, parameter2, ...* – параметри методу, які можуть передаватися методу для обробки.

Наприклад, потрібно створити метод *display_info* для класу *Person*, який виводить інформацію про об'єкт класу *Person*:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Створення екземпляра класу Person
person1 = Person("John", 30)

# Виклик методу display_info для об'єкта person1
person1.display_info()  # Виведе: Name: John, Age: 30
```

У цьому прикладі метод *display_info* визначений для класу *Person*. Він приймає параметр *self*, який представляє поточний об’єкт класу *Person*, та виводить інформацію про ім’я та вік об’єкта. Цей метод може бути викликаний для будь-якого об’єкта класу *Person*, і він буде виводити інформацію про цей об’єкт.

19.5. Застосування змінної self для доступу до атрибутів та методів усередині класу

Змінна *self* у Python використовується для доступу до атрибутів та методів класу всередині його власних методів. Коли метод класу викликається для конкретного об’єкта класу, цей об’єкт передається як перший аргумент методу й автоматично прив’язується до параметра *self*.

1. Доступ до атрибутів за допомогою *self*:

Під час доступу до атрибутів у межах методів класу використовується ключове слово *self*, за яким слідує ім’я атрибута.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Створення екземпляра класу Person
person1 = Person("John", 30)

# Доступ до атрибута name за допомогою self
print(person1.name)  # Виведе: John
```

2. Виклик методів за допомогою *self*:

Аналогічно, коли ви викликаєте метод класу всередині іншого методу класу, ви вказуєте *self* як перший аргумент методу, щоб звернутися до цього об’єкта.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

    def birthday(self):
        self.age += 1  # Збільшення віку на 1

# Створення екземпляра класу Person
person1 = Person("John", 30)

# Виклик методу birthday для об'єкта person1
person1.birthday()

# Виведення зміненого віку за допомогою методу display_info
person1.display_info()  # Виведе: Name: John, Age: 31
```

У цьому прикладі метод *birthday* збільшує вік об’єкта класу *Person* на 1. Щоб отримати доступ до атрибута *age* всередині методу *birthday*, ми використовуємо *self*.

19.6. Методи __init__() для задавання початкових значень атрибутам екземпляра класу

Метод *__init__()* у Python є спеціальним методом, який автоматично викликається під час створення нового екземпляра класу. Він використовується для ініціалізації початкових значень атрибутів об’єкта класу.

1. Клас з одним атрибутом:

```
class Person:
    def __init__(self, name):
        self.name = name

# Створення екземпляра класу Person з ім'ям "John"
person1 = Person("John")
print(person1.name)  # Виведе: John
```

2. Клас з декількома атрибутами:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# Створення екземпляра класу Point з координатами (2, 3)
point1 = Point(2, 3)
print(point1.x, point1.y)  # Виведе: 2 3
```

3. Клас з атрибутами за замовчуванням:

```
class Car:
    def __init__(self, make="Toyota", model="Camry"):
        self.make = make
        self.model = model

# Створення екземпляра класу Car зі значеннями за замовчуванням
car1 = Car()
print(car1.make, car1.model)  # Виведе: Toyota Camry

# Створення екземпляра класу Car з власними значеннями
car2 = Car("Honda", "Civic")
print(car2.make, car2.model)  # Виведе: Honda Civic
```

4. Клас зі списком атрибутів:

```
class Dog:
    def __init__(self, name, age, breed):
        self.name = name
        self.age = age
        self.breed = breed

# Створення екземпляра класу Dog з різними атрибутами
dog1 = Dog("Buddy", 5, "Labrador")
print(dog1.name, dog1.age, dog1.breed)  # Виведе: Buddy 5 Labrador
```

У кожному з цих прикладів метод *__init__()* приймає параметри, які використовуються для ініціалізації атрибутів об'єкта класу. Під час створення нового екземпляра класу ці параметри передаються до методу *__init__()* і використовуються для ініціалізації атрибутів об'єкта.

Документування класу і методів у Python. Документування класу і методів може бути виконане за допомогою коментарів у спеціальному форматі, такому як *docstrings*, а також за допомогою анотацій типів для атрибутів та параметрів методів.

1. Документування класу і методів за допомогою *docstrings*:

```
class Calculator:
    """Цей клас представляє простий калькулятор."""

    def __init__(self, value):
        """Ініціалізує калькулятор з вказаним значенням."""
        self.value = value

    def add(self, num):
        """Додає число до поточного значення калькулятора."""
        self.value += num
```

2. Анотація типів для атрибутів класу та параметрів методів:

```
class Rectangle:
    width: float
    height: float

    def __init__(self, width: float, height: float):
        self.width = width
        self.height = height

    def area(self, scale: float) -> float:
        return self.width * self.height * scale
```

3. Документування атрибутів та методів за допомогою docstrings:

```
class Employee:
    """Представляє працівника компанії."""

    name: str
    age: int

    def __init__(self, name: str, age: int):
        """Ініціалізує об'єкт класу Employee з ім'ям та віком."""
        self.name = name
        self.age = age

    def introduce(self) -> str:
        """Повертає рядок з представленням працівника."""
        return f"Привіт, мене звуть {self.name} і мені {self.age} років."
```

4. Анотація типів для параметрів методів та поверненого значення:

```
class Product:
    name: str
    price: float

    def __init__(self, name: str, price: float):
        self.name = name
        self.price = price

    def discounted_price(self, discount: float) -> float:
        """Обчислює знижену ціну товару з вказаним відсотком знижки."""
        return self.price * (1 - discount / 100)
```

5. Перегляд усіх атрибутів класу за допомогою *__dict__*:

```
class MyClass:
    name = "John"
    age = 30

print(MyClass.__dict__)
# Виведе: {'__module__': '__main__', 'name': 'John', 'age': 30, ...}
```

Метод *__dict__* класу дає змогу отримати словник, що містить усі атрибути класу та їх значення.

19.7. Метод __del__() як деструктор, що викликається під час видалення екземпляра класу

Метод *__del__()* у Python є спеціальним методом, який викликається автоматично, коли екземпляр класу видаляється з пам’яті або збирається збирач сміття (*garbage collector*). Використовуючи цей метод, ви можете визначити певні дії, які мають виконуватися перед тим, як об’єкт буде видалено з пам’яті.

```
class MyClass:
    def __init__(self, name):
        self.name = name

    def __del__(self):
        print(f"Екземпляр класу {self.name} видалений з пам'яті")

# Створення екземпляра класу
obj1 = MyClass("obj1")

# Видалення екземпляра класу
del obj1
```

Під час видалення екземпляра класу *obj1* метод *__del__()* буде викликаний автоматично і виведе повідомлення *"Екземпляр класу obj1 видалений з пам'яті"*.

Важливою річчю є те, що виклик цього методу не завжди гарантується, особливо якщо існують циклічні посилання між об'єктами. Такі ситуації можуть призвести до того, що збірник сміття не зможе правильно видалити об'єкти і метод *__del__()* не буде викликаний. Тому рекомендовано уникати залежностей між об'єктами та ресурсами поза Python (наприклад, відкриті файли) у методі *__del__()* і використовувати конструкцію *with* або інші засоби управління ресурсами.

Перелік питань для самоконтролю

1. Основні означення, що пов'язані з об'єктно-орієнтованим програмуванням.
2. Визначення класу й створення екземпляра класу.
3. Синтаксис створення атрибута класу.
4. Спосіб створення методу класу за допомогою інструкції **def.**
5. Застосування змінної **self** для доступу до атрибутів та методів всередині класу.

6. Методи **__init__()** для задавання початкових значень атрибутам екземпляра класу.

7. Метод **__del__()** як деструктор, що викликається під час видалення екземпляра класу.

8. Гетери та сетери.

9. Декоратор **@property.**

Рекомендовані джерела:

Основні: 1; 2; 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Створіть клас *Car*, який має атрибути *make* (марка) і *model* (модель). Додайте метод *display_info()*, який виводить інформацію про автомобіль у вигляді рядка "Make: [make], Model: [model]". Створіть об’єкт цього класу та викличте метод.

2. Створіть клас *Person*, який має атрибути класу *species* та атрибути екземпляра *name* і *age*. Встановіть *species* за замовчуванням на "Homo sapiens". Додайте метод `greet()`, який виводить привітання у форматі "Hello, my name is [name] and I am [age] years old". Створіть кілька об’єктів цього класу та викличте метод для кожного з них.

3. Створіть клас *Counter*, який має атрибут класу *count* (з початковим значенням 0). Додайте метод класу *increment()* для збільшення значення *count* на 1 і метод `get_count()` для отримання поточного значення `count`. Створіть кілька об’єктів цього класу і використовуйте методи для перевірки значень.

4. Створіть клас `MathUtils`, який має метод `add(a, b)`, що повертає суму двох чисел. Створіть об’єкт цього класу та викличте метод.

5. Створіть клас `Library`, який має атрибути екземпляра name і `location` та атрибут класу `book_count` (з початковим значенням 0). Додайте метод `add_book()` для збільшення `book_count` на 1 і метод `display_info()` для виведення інформації про бібліотеку. Створіть кілька об’єктів цього класу, додайте книги до кожної бібліотеки та виведіть інформацію про кожну бібліотеку.

6. Створіть клас `Book`, який має атрибути `title` і `author`. Визначте метод `__init__()`, який приймає ці параметри та задає початкові значення для атрибутів. Створіть об’єкт класу `Book` та виведіть його атрибути.

7. Створіть клас Car, який має атрибути make, model, і `year`. Визначте метод `__init__()` з параметрами за замовчуванням для `year`. Якщо рік не заданий, встановіть його на поточний рік. Створіть кілька об’єктів цього класу, вказавши або не вказавши рік.

8. Створіть клас `Student`, який має атрибути name, age, і `grades` (список оцінок). Визначте метод `__init__()` для ініціалізації цих атрибутів. Додайте метод `average_grade()`, який повертає середню оцінку студента. Створіть об’єкт цього класу та виведіть середню оцінку.

9. Створіть клас `Rectangle`, який має атрибути `width` і `height`. Визначте метод `__init__()` з перевіркою, щоб забезпечити, що обидва значення є додатніми числами. Якщо значення не відповідають умовам, піднімайте `ValueError`. Створіть кілька об’єктів цього класу, перевірте поведінку в разі неправильних значень.

10. Створіть клас `BankAccount`, який має атрибути класу `interest_rate` (ставка відсотка) і атрибути екземпляра `balance` (баланс). Визначте метод `__init__()` для ініціалізації балансу та метод `apply_interest()` для застосування відсотків до балансу. Створіть кілька об’єктів цього класу та застосуйте відсотки до кожного з них.

11. Створіть клас `Resource`, який має метод `__del__()`, що виводить повідомлення про те, що об’єкт був видалений. Створіть об’єкт цього класу та перевірте, чи викликається метод `__del__()` у разі видалення об’єкта.

12. Створіть клас `FileHandler`, який відкриває файл у режимі запису під час створення об’єкта та закриває його в методі `__del__()`. Перевірте, чи файл закривається автоматично під час видалення об’єкта.

13. Створіть клас `NetworkConnection`, який має атрибут для зберігання стану з’єднання. Додайте метод `__del__()`, який спробує закрити з’єднання. Додайте обробку помилок у `__del__()` для випадку, коли з’єднання вже закрите або не існує.

14. Створіть два класи `A` і `B`, де `B` має атрибут `a_instance`, який є екземпляром класу `A`. Додайте метод `__del__()` у обидва класи для виведення повідомлень про видалення. Переконайтеся, що `__del__()` у класі `B` викликається після видалення екземпляра `A`.

15. Створіть клас `ConnectionManager`, який управляє зовнішнім ресурсом, наприклад, відкриттям та закриттям файлу. Реалізуйте методи `__enter__()` і `__exit__()` для використання класу як контекстного менеджера. Перевірте, що метод `__del__()` також викликається, якщо об’єкт виходить з області видимості до завершення блоку `with`.

Тема 20
Інкапсуляція. Статичні методи, поля та методи класу

20.1. Ідея інкапсуляції.

20.2. Інкапсуляція та приховування даних.

20.3. Статичні методи, поля та методи класу. Методи класу.

20.4. Гетери та сетери. Використання декораторів @property, @_.setter.

20.1. Ідея інкапсуляції

Інкапсуляція – це один з основних принципів об'єктно-орієнтованого програмування, що полягає в обмеженні доступу до певних даних та функціональності об'єкта ззовні класу, а також в об'єднанні даних і методів, які з ними працюють, в одному компоненті. У Python інкапсуляція реалізується за допомогою модифікаторів доступу і методів класу.

Основні ідеї інкапсуляції:

1. Приховування даних. Змінні та методи можуть бути приховані від зовнішнього світу, а доступ до них здійснюється лише через публічні методи.

2. Захист даних. Доступ до даних може бути контрольований, щоб уникнути неправильного використання або змін.

3. Спрощення інтерфейсу. Внутрішня реалізація об'єкта може бути складною, але зовнішній інтерфейс до неї може бути простим і зрозумілим.

Реалізація інкапсуляції в Python за допомогою методів класу:

1. Використання приватних змінних. У Python немає справжніх приватних змінних, але є конвенція використання підкреслення перед ім'ям змінної для позначення приватності. Наприклад, *_variable*.

2. Публічні методи для доступу до приватних змінних: Клас може надавати публічні методи, які дають змогу отримати доступ до приватних змінних та виконати операції над ними. Ці методи називаються гетерами та сетерами.

3. Приховування методів: Клас може приховати певні методи, які не повинні бути використані ззовні. Це допомагає уникнути неналежного використання або зміни внутрішньої реалізації.

Ось приклад реалізації інкапсуляції в Python за допомогою методів класу:

```
class Car:
    def __init__(self, make, model, year):
        self._make = make  # Приватне поле
        self._model = model
        self._year = year

    def get_make(self):
        return self._make

    def set_make(self, make):
        self._make = make

    def get_model(self):
        return self._model

    def set_model(self, model):
        self._model = model

    def get_year(self):
        return self._year

    def set_year(self, year):
        self._year = year

# Створення екземпляра класу Car
my_car = Car("Toyota", "Camry", 2020)

# Зміна значення приватного поля через сетер
my_car.set_make("Honda")

# Отримання значення приватного поля через гетер
print(my_car.get_make())  # Виведе: Honda
```

У цьому прикладі дані про автомобіль (марка, модель, рік) приховані від зовнішнього світу за допомогою приватних змінних *_make*, *_model* та *_year*, і доступ до них здійснюється через гетери та сетери. Це забезпечує контрольований доступ до даних та забезпечує інкапсуляцію.

20.2. Інкапсуляція та приховування даних

Інкапсуляція та приховування даних – це два поняття, які часто використовуються в ООП, особливо в мові програмування Python.

Інкапсуляція – це механізм, який об'єднує дані та методи, які працюють з цими даними, в одному компоненті (класі) та захищає їх від зовнішнього доступу або маніпуляцій. Іншими словами, інкапсуляція дозволяє обмежити доступ до певних елементів класу, щоб забезпечити безпеку та правильне використання даних.

Приховування даних – це концепція, яка полягає в тому, що деякі дані або деталі реалізації класу можуть бути приховані від користувача класу. Це означає, що змінні та методи можуть бути оголошені як приватні, щоб уникнути прямого доступу до них ззовні класу.

У Python приховування даних не є суворим, так як в інших мовах програмування, таких як Java або C++, оскільки Python не має справжнього механізму приватності. Однак використання певних конвенцій та механізмів, таких як префікси підкреслення для позначення приватних змінних та методів, може допомогти приховати деякі елементи класу від користувача класу.

1. Приватні та захищені змінні класу:

- приватні змінні: вони позначаються префіксом подвійного підкреслення (`__`). Ці змінні не можуть бути доступні за межами класу або його підкласів;

- захищені змінні: вони позначаються одним підкресленням (`_`) на початку імені. Ці змінні мають бути розглянуті як приватні, але їх можна отримати доступ до класів, що успадковуються від цього класу.

2. Приватні та захищені методи класу:

- приватні методи: вони також позначаються префіксом подвійного підкреслення (`__`). Ці методи не можуть бути викликані за межами класу, в якому вони оголошені;

- захищені методи: вони також позначаються одним підкресленням (`_`) на початку імені. Ці методи розглядаються як приватні, але можуть бути успадковані підкласами та викликані з них.

```
class MyClass:
    def __init__(self):
        self.__private_var = 10  # Приватна змінна
        self._protected_var = 20  # Захищена змінна

    def __private_method(self):
        print("Це приватний метод")

    def _protected_method(self):
        print("Це захищений метод")

    def access_private(self):
        print(self.__private_var)  # Метод класу може отримати доступ до приватної змінної

    def access_protected(self):
        print(self._protected_var)  # Метод класу може отримати доступ до захищеної змінної

# Створення екземпляра класу
obj = MyClass()

# Намагаємося звернутися до приватних та захищених змінних поза класом
print(obj._MyClass__private_var)  # Виведе: 10
print(obj._protected_var)  # Виведе: 20

# Виклик приватного та захищеного методів поза класом
obj._MyClass__private_method()  # Виведе: Це приватний метод
obj._protected_method()  # Виведе: Це захищений метод
```

У цьому прикладі *__private_var* є приватною змінною, а *_protected_var* – захищеною. Аналогічно, *__private_method()* є приватним методом, а *_protected_method()* – захищеним.

20.3. Статичні методи, поля та методи класу. Методи класу

Статичні методи, поля та методи класу – це концепції, які дають змогу працювати з даними та функціональністю класу, не створюючи екземпляра класу. Вони є корисними для групування функцій і даних, які пов'язані з класом, але не потребують доступу до екземпляра класу.

1. Статичні методи:

```
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def subtract(x, y):
        return x - y

# Виклик статичних методів без створення екземпляра класу
print(MathOperations.add(5, 3))  # Виведе: 8
print(MathOperations.subtract(10, 4))  # Виведе: 6
```

2. Поля та методи класу:

```
class Dog:
    count = 0  # Поле класу для підрахунку кількості створених екземплярів

    def __init__(self, name):
        self.name = name
        Dog.count += 1  # Збільшуємо лічильник під час створення кожного екземпляра
```

```
    @classmethod
    def get_count(cls):
        return cls.count

# Створення екземплярів класу
dog1 = Dog("Buddy")
dog2 = Dog("Max")

# Виклик методу класу для отримання кількості створених екземплярів
print(Dog.get_count())  # Виведе: 2
```

3. Статичні методи та методи класу в поєднанні:

```
class StringUtils:
    @staticmethod
    def reverse_string(string):
        return string[::-1]

    @classmethod
    def capitalize_string(cls, string):
        return cls.reverse_string(string).capitalize()

# Виклик статичного методу
print(StringUtils.reverse_string("hello"))  # Виведе: olleh

# Виклик методу класу через екземпляр класу
print(StringUtils.capitalize_string("hello"))  # Виведе: Olleh
```

4. Статичні методи для функцій-утиліт:

```
class UtilityFunctions:
    @staticmethod
    def is_prime(num):
        if num <= 1:
            return False
        for i in range(2, int(num  0.5) + 1):
            if num % i == 0:
                return False
        return True

# Виклик статичного методу для перевірки, чи число просте
print(UtilityFunctions.is_prime(7))  # Виведе: True
```

У всіх цих прикладах методи класу та статичні методи використовуються для організації функціональності, що відноситься до класу, без необхідності створення екземплярів класу.

20.4. Гетери та сетери. Використання декораторів @property, @_.setter

Гетери та сетери – це методи, які використовуються для отримання (гетери) та встановлення (сетери) значень приватних атрибутів класу. Гетери та сетери дають змогу контролювати доступ до даних та виконувати додаткові дії під час їхнього читання або запису.

1. Простий гетер та сетер:

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height
```

```
    def get_width(self):
        return self._width

    def set_width(self, width):
        if width > 0:
            self._width = width

    def get_height(self):
        return self._height

    def set_height(self, height):
        if height > 0:
            self._height = height

# Створення екземпляра класу Rectangle
rect = Rectangle(10, 5)

# Використання гетера та сетера
print(rect.get_width())  # Виведе: 10
rect.set_width(20)
print(rect.get_width())  # Виведе: 20
```

2. Використання декораторів *@property* для створення гетера та сетера:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius
```

```
    @radius.setter
    def radius(self, radius):
        if radius > 0:
            self._radius = radius

# Створення екземпляра класу Circle
circle = Circle(5)

# Використання гетера та сетера
print(circle.radius)  # Виведе: 5
circle.radius = 10
print(circle.radius)  # Виведе: 10
```

3. Використання декораторів *@property* без сетера (тільки гетер):

```
class Square:
    def __init__(self, side):
        self._side = side

    @property
    def side(self):
        return self._side

# Створення екземпляра класу Square
square = Square(4)

# Використання гетера
print(square.side)  # Виведе: 4

# У разі спроби змінити значення side без сетера виникне AttributeError
# square.side = 5  # AttributeError: can't set attribute
```

4. Використання гетера та сетера з додатковими перевірками:

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, celsius):
        if celsius < -273.15:
            raise ValueError("Temperature cannot be below -273.15°C")
        self._celsius = celsius

# Створення екземпляра класу Temperature
temp = Temperature(25)

# Використання гетера та сетера з додатковими перевірками
print(temp.celsius)  # Виведе: 25
temp.celsius = 30
print(temp.celsius)  # Виведе: 30

# У разі спроби встановити неприпустиме значення виникне ValueError
# temp.celsius = -300  # ValueError: Temperature cannot be below -273.15°C
```

5. Використання гетера для обчислення значення на основі інших атрибутів:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius
```

```
    @property
    def area(self):
        return 3.14 * self._radius  2

# Створення екземпляра класу Circle
circle = Circle(5)

# Використання гетера для обчислення площі кола
print(circle.area)  # Виведе: 78.5
```

Декоратори *@property* та *@.setter* використовуються для створення властивостей класу, які можна читати та записувати. Вони дають змогу створити інтерфейс доступу до приватних або захищених змінних класу без прямого доступу до них.

1. Використання *@property* для створення властивості.

У Python декоратор *@property* дає змогу створювати властивості (*properties*) в класах. Це дає змогу визначати методи, які можна використовувати як атрибути, надаючи можливість контролювати доступ до даних (зчитування, запис, видалення) в класі більш елегантно й зрозуміло.

Основи використання *@property*.

Декоратор *@property* перетворює метод на властивість, яку можна використовувати так само, як звичайний атрибут, але водночас дає змогу виконувати певну логіку під час зчитування або запису значення.

Базовий приклад використання *@property*:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius
```

```
    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative.")
        self._radius = value

    @property
    def area(self):
        import math
        return math.pi * (self._radius ** 2)
```

Пояснення

1. Оголошення властивості з *@property*. Декоратор *@property* використовується для визначення методів, які будуть доступні як властивості класу. У цьому випадку метод *radius* стає властивістю.

2. Гетер (*getter*). Метод, помічений декоратором *@property*, зазвичай використовується для отримання значення атрибута. У прикладі вище *radius* є Гетером.

3. Сетер (*setter*). Для створення властивості, яку можна змінювати, використовується декоратор *@<property_name>.setter*. Це дає змогу визначити, як потрібно обробляти запис значення. У прикладі вище *radius* є Сетером для атрибута *_radius*.

4. Делітер (*deleter*). Можна також визначити метод для видалення значення властивості за допомогою декоратора *@<property_name>.deleter*.

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius
```

```
    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative.")
        self._radius = value

    @radius.deleter
    def radius(self):
        print("Deleting radius...")
        del self._radius

    @property
    def area(self):
        import math
        return math.pi * (self._radius ** 2)
```

Приклад використання

```python
# Створення об’єкта Circle
circle = Circle(5)

# Отримання значення властивості
print(circle.radius)  # Виведе: 5

# Зміна значення властивості
circle.radius = 10
print(circle.radius)  # Виведе: 10

# Отримання значення іншої властивості
print(circle.area)  # Виведе: 314.159...

# Видалення властивості
del circle.radius
```

1. Переваги використання *@property*:

- інтерфейс. Дає змогу використовувати методи як атрибути, що робить код більш чистим і зрозумілим;

- контроль доступу. Дає змогу реалізувати логіку під час доступу до атрибутів (наприклад, валідація значень під час запису);

- інкапсуляція. Зменшує прямий доступ до атрибутів, надаючи контрольовану точку доступу через властивості.

Декоратор *@property* є потужним інструментом для покращення інтерфейсу класів і дає змогу легко реалізувати контроль над даними без необхідності створювати явні методи для отримання і встановлення значень.

2. Використання *@property* для обчислюваних властивостей:

У Python декоратор *@property* дає змогу створювати обчислювані властивості в класах. Обчислювальні властивості – це такі властивості, які не зберігають значення безпосередньо, а обчислюють його динамічно на основі інших атрибутів або зовнішніх чинників. Це може бути корисно для виконання обчислень або для отримання значень, що змінюються на основі стану об’єкта.

Основи використання *@property* для обчислюваних властивостей.

1. Оголошення обчислювальної властивості. Використовується *@property* для визначення методу, який буде обчислювати й повертати значення властивості.

2. Зчитування значення. Коли властивість доступна, буде викликатися метод, помічений *@property*.

3. Не потрібно визначати Сетер або делеттер: Для обчислювальних властивостей не потрібно визначати методи для запису або видалення значення, якщо ви не плануєте змінювати властивість.

```
# Обчислювана площа кола
import math

class Circle:
    def __init__(self, radius):
        self._radius = radius
```

```
    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative.")
        self._radius = value

    @property
    def area(self):
        return math.pi * (self._radius ** 2)
```

У цьому прикладі *area* є обчислювальною властивістю. Її значення обчислюється на основі поточного значення атрибута *_radius*. Ви не зберігаєте площу як атрибут; натомість вона розраховується під час кожного доступу.

```
# Обчислення гіпотенузи трикутника
class Triangle:
    def __init__(self, base, height):
        self._base = base
        self._height = height

    @property
    def base(self):
        return self._base

    @base.setter
    def base(self, value):
        if value < 0:
            raise ValueError("Base cannot be negative.")
        self._base = value
```

```
    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        if value < 0:
            raise ValueError("Height cannot be negative.")
        self._height = value

    @property
    def hypotenuse(self):
        return (self._base ** 2 + self._height ** 2) ** 0.5

    @property
    def area(self):
        return 0.5 * self._base * self._height
```

Тут *hypotenuse* та *area* є обчислювальними властивостями. *hypotenuse* обчислюється на основі значень *base* і *height*, а *area* обчислюється на основі *base* і *height*.

```
# Використання властивостей
# Створення об'єкта Circle
circle = Circle(5)
print(circle.radius)  # Виведе: 5
print(circle.area)    # Виведе: 78.53981633974483 (площа кола)

# Зміна значення властивості
circle.radius = 10
print(circle.area)    # Виведе: 314.1592653589793

# Створення об'єкта Triangle
triangle = Triangle(3, 4)
print(triangle.base)      # Виведе: 3
print(triangle.height)    # Виведе: 4
print(triangle.hypotenuse) # Виведе: 5.0
print(triangle.area)      # Виведе: 6.0
```

Переваги обчислювальних властивостей:

- економія пам’яті – не потрібно зберігати додаткові атрибути для результатів обчислень;

- динамічність – значення властивостей завжди актуальні та відповідають потньому стану об’єкта;

- інтерфейс – легше використовувати методи як властивості, що робить код більш читабельним і зручним.

Обчислювальні властивості забезпечують гнучкість у створенні класів, даючи змогу виконувати складні обчислення без необхідності зберігати додаткові дані.

3. Використання *@.setter* для встановлення значення властивості.

У Python декоратор *@property* дає змогу створювати властивості, а *@<property_name>.setter* – це декоратор, що дає змогу визначити метод для встановлення значення цієї властивості. Це корисно, коли ви хочете контролювати, як встановлюються значення для певних атрибутів класу, включаючи перевірку коректності значень або виконання додаткових дій під час запису значення.

Основи використання *@<property_name>.setter*.

1. Оголошення властивості. Спочатку використовується декоратор *@property* для створення Гетера – методу, який дає змогу отримувати значення властивості.

2. Оголошення Сетера. Для визначення методу, що буде використовуватися для встановлення значення властивості, використовують декоратор *@<property_name>.setter*, де *<property_name>* – це ім’я властивості.

```
# Синтаксис
class MyClass:
    def __init__(self, value):
        self._value = value

    @property
    def value(self):
        return self._value
```

```
    @value.setter
    def value(self, new_value):
        # Додаткова логіка для встановлення значення
        if new_value < 0:
            raise ValueError("Value cannot be negative.")
        self._value = new_value
```

Розглянемо клас *Person*, де властивість *age* має обмеження, щоб не давати можливості встановлювати від'ємні значення.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value < 0:
            raise ValueError("Age cannot be negative.")
        self._age = value

    @property
    def is_adult(self):
        return self._age >= 18
```

1. Гетер *age*. Декоратор *@property* перетворює метод *age* на властивість, яка дає змогу отримувати значення *_age*.

2. Сетер *age*. Декоратор *@age.setter* визначає метод для встановлення значення властивості *age*. Можна реалізувати валідацію значення або інші перевірки перед тим, як встановити нове значення.

3. Обчислювальна властивість *is_adult*. Не має Сетера і лише повертає результат обчислення на основі значення *age*.

```
# Використання класу

person = Person("Alice", 30)

# Отримання значення властивості
print(person.age)     # Виведе: 30

# Зміна значення властивості
person.age = 35
print(person.age)     # Виведе: 35

# Спроба встановлення від'ємного віку (викине виключення)
try:
    person.age = -5
except ValueError as e:
    print(e)          # Виведе: Age cannot be negative.

# Перевірка обчислювальної властивості
print(person.is_adult) # Виведе: True
```

Переваги використання *@<property_name>.setter*.

- контроль доступу – можливість контролювати, як значення встановлюються для атрибутів класу;

- валідація даних – легкість у перевірці даних перед їх установленням;

- чистота інтерфейсу – можливість використовувати властивості так само, як звичайні атрибути, з одночасним контролем над їх установленням.

Використання *@property* і *@<property_name>.setter* робить ваш код більш модульним і зрозумілим, даючи змогу легко реалізовувати контроль над даними в класах.

4. Використання *@property* для створення тільки читабельної властивості.

У Python декоратор *@property* дає змогу створювати тільки читабельні властивості. Це особливо корисно, коли потрібно надати доступ до даних класу для читання, але заборонити їх зміну ззовні класу. Такі властивості не мають сетера, і тому значення можна тільки отримувати, але не змінювати.

Основи створення тільки читабельних властивостей:

1. оголошення Гетера – декоратор *@property* використовується для створення методу, який повертає значення властивості;

2. відсутність Сетера – якщо ви не визначите метод з декоратором *@<property_name>.setter*, властивість стане тільки читабельною.

```
# Синтаксис
class MyClass:
    def __init__(self, value):
        self._value = value

    @property
    def value(self):
        return self._value
```

Розглянемо клас *Rectangle*, де властивість *area* обчислюється на основі ширини та висоти прямокутника і є тільки читабельною.

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    @property
    def width(self):
        return self._width
```

```
    @width.setter
    def width(self, value):
        if value < 0:
            raise ValueError("Width cannot be negative.")
        self._width = value

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        if value < 0:
            raise ValueError("Height cannot be negative.")
        self._height = value

    @property
    def area(self):
        return self._width * self._height
```

Пояснення:

1. Гетер *width* і *height* – властивості *width і height* мають як Гетери, так і Сетери, щоб дозволити читання і зміну значень.

2. Тільки читабельна властивість *area* – властивість *area* має лише Гетер, який обчислює площу на основі ширини і висоти. Вона не має Сетера, тому значення властивості *area* не можна змінювати ззовні класу.

```
# Використання класу
# Створення об'єкта Rectangle
rect = Rectangle(5, 10)

# Отримання значення властивостей
print(rect.width)    # Виведе: 5
print(rect.height)   # Виведе: 10
print(rect.area)     # Виведе: 50
```

```
    # Зміна значень властивостей
    rect.width = 7
    rect.height = 12
    print(rect.area)    # Виведе: 84

    # Спроба встановлення значення для властивості `area` (викине виключення)
    try:
        rect.area = 100
    except AttributeError as e:
        print(e)       # Виведе: can't set attribute
```

Переваги використання тільки читабельних властивостей.

- Захист даних. Забезпечує захист даних від змін ззовні класу, що дає змогу уникнути несанкціонованої або некоректної зміни стану об'єкта.

- Інтерфейс. Забезпечує зрозумілий і зрозумілий інтерфейс для доступу до обчислювальних або агрегованих значень.

- Інкапсуляція. Допомагає реалізувати інкапсуляцію, надаючи тільки необхідний рівень доступу до внутрішніх даних об'єкта.

Використання *@property* для створення тільки читабельних властивостей дає змогу вам контролювати, як дані отримуються і змінюються, покращуючи надійність та підтримуваність вашого коду.

Перелік питань для самоконтролю

1. Ідея інкапсуляції та її реалізація в Python за допомогою методів класу.
2. Інкапсуляція та приховування даних.
3. Статичні методи, поля та методи класу.
4. Відношення між сутностями.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Створіть клас Person з приватними атрибутами `__name` і `__age`. Реалізуйте публічні методи для отримання і встановлення значень цих атрибутів. Перевірте роботу класу, створивши об’єкт і змінюючи його атрибути через публічні методи.

2. Створіть клас `Account`, у якого є приватний атрибут `__balance`. Додайте методи для внесення і зняття грошей з рахунку, а також метод для перевірки балансу. Переконайтеся, що баланси можна змінювати лише через публічні методи.

3. Створіть клас `MathUtils` зі статичним методом `add(a, b)`, який повертає суму двох чисел. Використовуйте цей метод без створення екземпляра класу.

4. Створіть клас `Library`, який має статичне поле `total_books` для підрахунку загальної кількості книг. Додайте методи для додавання і видалення книг, які оновлюють це поле. Перевірте роботу класу, створюючи об’єкти і використовуючи методи.

5. Створіть клас `Student` з атрибутом класу `school_name`. Реалізуйте метод класу `change_school_name()` для зміни значення `school_name` і метод `get_school_name()` для отримання значення. Створіть кілька об’єктів цього класу і перевірте, як зміна значення через метод класу впливає на всі екземпляри.

6. Створіть клас `Validator`, який має статичний метод `is_positive_integer(value)`, який перевіряє, чи є значення додатнім цілим числом. Використовуйте цей метод для перевірки значень перед створенням об’єктів класу.

7. Створіть клас `Circle`, який має статичне поле `pi` для значення числа π. Додайте метод для обчислення площі кола, використовуючи `pi`. Встановіть і змініть значення `pi` через клас.

8. Створіть клас `Date` з атрибутами `day`, `month`, `year`. Реалізуйте метод класу `from_string(date_string)` для створення об’єкта з рядка у форматі "dd-mm-yyyy". Додайте метод для виведення дати у форматі "dd Month yyyy".

9. Створіть базовий клас `Vehicle` з приватними атрибутами `__make` і `__model`. Створіть похідний клас Car, який додає атрибут `__year` і метод для виведення повної інформації про автомобіль. Переконайтеся, що атрибути базового класу недоступні напряму в похідному класі.

10. Створіть клас `Temperature` з приватним атрибутом `__celsius`. Додайте властивості для доступу до значення у градусах Цельсія і Фаренгейта, де конвертація між шкалами здійснюється через методи властивостей.

11. Створіть клас `StringUtils` зі статичним методом `reverse_string(s)`, який повертає реверсований рядок. Використовуйте цей метод без створення екземпляра класу.

12. Створіть клас `Counter`, який має атрибут класу `count` для підрахунку кількості створених екземплярів. Реалізуйте метод класу `get_count()` для отримання значення `count`.

13. Створіть клас `Statistics`, який має статичний метод `mean(numbers)`, що обчислює середнє арифметичне з переданого списку чисел. Використовуйте цей метод без створення об’єкта класу.

14. Створіть клас `Company` з атрибутом класу `company_name`. Реалізуйте метод класу `get_company_name()` для отримання значення `company_name`. Додайте метод для зміни значення `company_name`.

15. Створіть клас `BankAccount` з приватними атрибутами `__balance` і `__account_number`. Реалізуйте методи для внесення і зняття грошей з рахунку з перевіркою валідності операцій.

16. Створіть клас `Config` зі статичним полем `settings`, яке зберігає налаштування конфігурації. Реалізуйте метод для оновлення конфігурацій і метод для отримання налаштувань.

17. Створіть клас Person з приватними атрибутами `__age` і `__name`. Реалізуйте методи для встановлення значення age, який має бути між 0 і 120, і name, який має бути непорожнім рядком.

18. Створіть клас `Rectangle` з атрибутами `width` і `height`. Реалізуйте метод класу `from_string(data)` для створення об’єкта з рядка у форматі "width,height".

19. Створіть клас `TemperatureConverter` з приватними атрибутами `__celsius` і `__fahrenheit`. Реалізуйте статичні методи для конвертації між шкалами і використовуйте ці методи у властивостях класу для доступу і зміни температури.

20. Створіть клас Person з приватними атрибутами `__name` і `__age`. Використовуйте декоратори `@property` і `@<property_name>.setter` для доступу та зміни цих атрибутів.

21. Створіть клас `Product` з приватним атрибутом `__price`. Використовуйте декоратори `@property` і `@price.setter` для доступу до `__price` та валідації його значення (має бути додатнім числом).

22. Створіть клас `Temperature` з приватним атрибутом `__celsius`. Використовуйте декоратори для перетворення значення між Цельсієм і Фаренгейтом через властивості.

23. Створіть клас Person з приватним атрибутом `__full_name`. Реалізуйте Гетер та Сетер для обробки атрибута, щоб `__full_name` завжди був у форматі "First Last".

24. Створіть клас `Rectangle` з приватними атрибутами `__width` і `__height`. Реалізуйте Гетери та Сетери для обчислення площі прямокутника.

25. Створіть клас `Temperature` з приватним атрибутом `__kelvin`. Реалізуйте Гетер та Сетер для конвертації значення з Кельвіна в Цельсій та Фаренгейт.

26. Створіть клас `Account` з приватним атрибутом `__balance`. Реалізуйте Гетер та Сетер для атрибута, щоб лише додатні значення могли бути присвоєні.

27. Створіть клас `Student` з приватним атрибутом `__age`. Реалізуйте Гетер та Сетер для віку, де вік має бути в межах від 18 до 100 років.

28. Створіть клас `Weight` з приватним атрибутом `__kilograms`. Реалізуйте Гетер та Сетер для конвертації значення з кілограмів у фунти і навпаки.

Тема 21
Спадкування

21.1. Відношення між сутностями.

21.2. Спадкування.

21.3. Поняття базового класу (суперкласу) та похідного класу (підкласу).

21.4. Функція super().

21.5. Множинне спадкування.

21.6. Асоціація, агрегація, композиція.

21.7. Написання класів-домішок.

21.1. Відношення між сутностями

У програмуванні відношення між сутностями описується як зв'язки, що визначають взаємодію між об'єктами або класами. Ці відношення допомагають структурувати програму та визначають поведінку об'єктів у системі.

1. Агрегація. Відношення, коли один об'єкт містить або включає в себе інший об'єкт, але обидва можуть існувати незалежно один від одного. Наприклад, клас «Команда» може містити список об'єктів класу «Гравець».

2. Композиція. Схожа на агрегацію, але з більш сильним зв'язком між об'єктами. Один об'єкт є частиною іншого об'єкта й не може існувати без нього. Наприклад, клас «Автомобіль» може мати об'єкт класу «Двигун», який є частиною автомобіля.

3. Спадкування (наслідування). Відношення, коли один клас може успадкувати властивості та методи іншого класу. Клас, що успадковує, відомий як підклас або дочірній клас, а клас, який надає свої властивості та методи, – як батьківський клас або суперклас. Це дає змогу створювати ієрархію класів та використовувати існуючий код.

4. Асоціація. Зв'язок між двома об'єктами, коли вони взаємодіють один з одним. Це може бути взаємна або одностороння взаємодія. Наприклад, клас «Клієнт» може асоціюватися з класом «Замовлення», коли клієнт розміщає замовлення.

5. Залежність. Коли зміна в одному класі впливає на інший клас. Зазвичай це виникає, коли один клас використовує інший клас як параметр у методі або як аргумент конструктора. Наприклад, клас «Автомобіль» може мати метод, який приймає об’єкт класу «Двигун».

Ці відношення між сутностями в програмуванні допомагають організувати код, створюючи моделі даних та визначаючи взаємодію між об’єктами в системі.

21.2. Спадкування

Успадкування в програмуванні, зокрема в мові Python, відноситься до механізму, за допомогою якого клас може успадковувати атрибути і методи від іншого класу, відомого як базовий клас або батьківський клас. Це дає змогу створювати ієрархії класів, де класи можуть спадковувати функціональність один від одного.

Припустимо, ми маємо базовий клас *Animal*, який містить деякі загальні властивості та методи для всіх тварин, і підклас *Dog*, який успадковує властивості та методи від класу *Animal* і може мати свої власні унікальні властивості та методи.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        pass  # Базова реалізація звуку для всіх тварин

class Dog(Animal):  # Dog успадковує Animal
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def make_sound(self):
        return "Woof!"  # Перевизначаємо метод make_sound для класу Dog
```

У цьому прикладі клас *Dog* успадковує клас *Animal*, тому всі атрибути й методи, які належать класу *Animal*, доступні у класі *Dog*. Ми використовуємо *super().__init__()* у конструкторі класу *Dog*, щоб викликати конструктор класу *Animal* і передати аргумент *name*.

Під час виклику методу *make_sound()* для об'єкта класу *Dog* Python використовує реалізацію методу з класу *Dog*, оскільки вона перевизначена в цьому підкласі. Якщо б ми не перевизначили метод *make_sound()* у класі *Dog*, він би успадковував базову реалізацію з класу *Animal*.

Це дає змогу створювати ієрархію класів, де класи надають узагальнені абстракції (наприклад, клас *Animal* з універсальними властивостями та методами для всіх тварин) і спеціалізовані класи (наприклад, клас *Dog* з унікальними властивостями та методами для собак).

21.3. Поняття базового класу (суперкласу) та похідного класу (підкласу)

У мові програмування Python базовий клас, який іноді також називають суперкласом, є класом, який надає загальні атрибути та методи, які можуть бути успадковані іншими класами. Похідний клас, який також відомий як підклас, успадковує цю функціональність від базового класу і може розширити його, додавши власні атрибути та методи або перевизначивши вже існуючі.

```
class Animal:  # Базовий клас (суперклас)
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        print("Some generic sound")

class Dog(Animal):  # Похідний клас (підклас)
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed
```

```
    def make_sound(self):
        print("Woof!")

class Cat(Animal):  # Інший похідний клас (підклас)
    def __init__(self, name, color):
        super().__init__(name)
        self.color = color

    def make_sound(self):
        print("Meow!")
```

У цьому прикладі *Animal* є базовим класом або суперкласом, оскільки він містить загальні атрибути та методи, які можуть бути успадковані іншими класами. Класи *Dog* та *Cat* є похідними класами або підкласами, оскільки вони успадковують функціональність від класу *Animal*.

Класи-підкласи можуть мати свої власні атрибути та методи, і вони можуть також перевизначати методи базового класу за допомогою того самого імені методу. Коли ви викликаєте метод для об'єкта класу-підкласу, Python спочатку шукає метод у самому класі-підкласі, а потім, якщо метод не знайдено, у його суперкласах. Це дає змогу зберігати ієрархічну структуру класів та зручно використовувати спільний код.

21.4. Функція super()

Функція *super()* у Python використовується для отримання доступу до методів та атрибутів батьківського класу в похідних класах. Вона дає змогу уникнути явного вказування імені батьківського класу і може бути корисною під час успадкування з багатьох рівнів.

1. Виклик методу батьківського класу з методу похідного класу:

```
class Parent:
    def hello(self):
        print("Hello from Parent")

class Child(Parent):
    def hello(self):
        super().hello()  # Виклик методу батьківського класу
        print("Hello from Child")

child = Child()
child.hello()
```

2. Виклик конструктора батьківського класу з конструктора похідного класу:

```
class Parent:
    def __init__(self, name):
        self.name = name

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Виклик конструктора батьківського класу
        self.age = age

child = Child("Alice", 5)
print(child.name)
print(child.age)
```

3. Виклик статичного методу батьківського класу з методу похідного класу:

```
class Parent:
    @staticmethod
    def static_method():
        print("Static method from Parent")

class Child(Parent):
    @staticmethod
    def static_method():
        super().static_method()
# Виклик статичного методу батьківського класу
        print("Static method from Child")

Child.static_method()
```

4. Виклик методу з іншого класу, який успадковує той самий клас, як і похідний клас:

```
class Base:
    def hello(self):
        print("Hello from Base")

class ChildA(Base):
    def hello(self):
        print("Hello from ChildA")

class ChildB(Base):
    def hello(self):
        super(ChildA, self).hello()  # Виклик методу з іншого класу
        print("Hello from ChildB")

child = ChildB()
child.hello()
```

5. Виклик методу з батьківського класу зовнішньо:

```
class Parent:
    def hello(self):
        print("Hello from Parent")

class Child(Parent):
    pass

child = Child()
super(Child, child).hello()  # Виклик методу батьківського класу ззовні
```

21.5. Множинне спадкування

Множинне спадкування в Python відбувається, коли клас успадковує функціональність від більше ніж одного базового класу. Це дає змогу створювати класи з комбінованою функціональністю з різних джерел. Однак множинне спадкування може призвести до складнощів, які пов’язані з конфліктами імен або неоднозначністю.

1. Простий приклад множинного спадкування:

```
class A:
    def method(self):
        print("Method from class A")

class B:
    def method(self):
        print("Method from class B")

class C(A, B):  # Множинне спадкування від класів A та B
    pass

obj = C()
obj.method()  # Викликає метод з класу A, оскільки він з'являється першим у списку успадкованих класів
```

У цьому прикладі клас *C* успадковує метод *method()* як від класу *A*, так і від класу *B*. Оскільки клас *A* міститься першим у списку успадкованих класів, метод *method()* з класу *A* буде використовуватися.

2. Конфлікт імен під час множинного спадкування:

```
class A:
    def method(self):
        print("Method from class A")

class B:
    def method(self):
        print("Method from class B")

class C(A, B):  # Множинне спадкування від класів A та B
    def method(self):  # Перевизначення методу method у класі C
        print("Method from class C")

obj = C()
obj.method()  # Викликає перевизначений метод з класу C
```

У цьому прикладі клас *C* успадковує метод *method()* як від класу *A*, так і від класу *B*. Однак у класі `C` метод *method()* перевизначений, тому використовується версія з класу *C*.

3. Множинне спадкування з додатковими параметрами:

```
class A:
    def method(self):
        print("Method from class A")

class B:
    def method(self):
        print("Method from class B")

class C(A, B):
    def method_C(self):  # Додатковий метод у класі C
        print("Additional method from class C")
```

```
obj = C()
obj.method()  # Викликає метод з класу A, оскільки він з'являється першим у списку успадкованих класів
obj.method_C()  # Викликає метод з класу C
```

У цьому прикладі клас *C* успадковує метод *method()* як від класу *A*, так і від класу *B*. Крім того, у класі *C* є додатковий метод *method_C()*. Об'єкти класу *C* можуть використовувати методи з обох батьківських класів та свої власні методи.

Множинне спадкування може бути корисним інструментом, але варто бути обережним під час його використання, щоб уникнути конфліктів імен або неоднозначностей у функціональності.

Під час множинного спадкування в Python можуть виникати деякі проблеми, такі як конфлікти імен методів або атрибутів, а також неоднозначність порядку розширення. Нижче наведені проблеми з поясненнями та способи їх усунення:

1. Конфлікти імен методів або атрибутів:

Проблема: якщо два або більше базових класів мають методи або атрибути з однаковими іменами, це призведе до конфлікту.

```
class A:
    def method(self):
        print("Method from class A")

class B:
    def method(self):
        print("Method from class B")

class C(A, B):
    pass

obj = C()
obj.method()  # Помилка: неоднозначність імен методів
```

Усунення: одним зі способів уникнути цієї проблеми є перейменування методів або атрибутів у класах-потомках або використання одного з методів за замовчуванням.

2. Неоднозначність порядку розширення:

Проблема: під час множинного спадкування може виникнути неоднозначність, якщо порядок, у якому вказані базові класи під час оголошення похідного класу, неоднозначний або конфліктний.

```
class A:
    def method(self):
        print("Method from class A")

class B(A):
    pass

class C(A):
    pass

class D(B, C):  # Множинне спадкування від класів B та C
    pass

obj = D()
obj.method()  # Помилка: неоднозначність порядку розширення
```

Усунення: один зі способів уникнути цієї проблеми полягає в явному вказанні потрібного порядку базових класів під час оголошення класу-потомка.

3. Використання абстрактних класів (або інтерфейсів):

Проблема: щоб уникнути конфліктів імен або неоднозначності, можна використовувати абстрактні класи (або інтерфейси), які визначають лише сигнатури методів, а потім у похідних класах реалізовують ці методи згідно з потребами.

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Абстрактний клас
    @abstractmethod
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")

dog = Dog()
dog.make_sound()

cat = Cat()
cat.make_sound()
```

У цьому прикладі *Animal* є абстрактним класом, який визначає метод *make_sound()* без реалізації. *Dog* та *Cat* успадковують цей клас і реалізовують метод *make_sound()* відповідно для собаки та кота.

Множинне спадкування може бути корисним інструментом, але варто усвідомлювати потенційні проблеми, які можуть виникнути, і використовувати відповідні стратегії для їх усунення.

21.6. Асоціація, агрегація, композиція

Асоціація, агрегація та композиція – це три різних типи взаємозв'язків між об'єктами в об'єктно-орієнтованому програмуванні. Кожен з цих типів має свої особливості і ступінь залежності між об'єктами.

1. Асоціація – це тип взаємозв'язку, коли об'єкти взаємодіють один з одним, але не є частиною життєвого циклу іншого об'єкта. Тобто об'єкти можуть взаємодіяти, але можуть існувати незалежно один від одного.

Приклад: Відносини між водієм та автомобілем є прикладом асоціації. Водій може взаємодіяти з автомобілем (повертати, керувати, гальмувати тощо), але автомобіль існує незалежно від водія.

2. Агрегація – це тип взаємозв'язку, коли один об'єкт є частиною іншого об'єкта, але може існувати самостійно від нього. У цьому випадку частина може належати багатьом цілим.

Приклад: Відносини між автомобілем та колесом є прикладом агрегації. Колесо є частиною автомобіля і може бути прив'язано до нього, але воно також може бути використане в інших автомобілях або існувати самостійно.

3. Композиція – це тип взаємозв'язку, коли один об'єкт є частиною іншого об'єкта і не може існувати без нього. Тобто частина належить лише одному цілому і життєвий цикл частини залежить від цілого.

Приклад: Відносини між автомобілем та двигуном є прикладом композиції. Двигун є необхідною частиною автомобіля і не може існувати без нього. Якщо автомобіль знищується, то й двигун також перестає існувати.

Усі ці типи взаємозв'язків мають свої особливості та використовуються в програмуванні залежно від потреб конкретної ситуації. Обережне використання цих концепцій допомагає створювати добре структурований і легко зрозумілий код.

1. Асоціація:

```
class Person:
    def __init__(self, name):
        self.name = name

class Company:
    def __init__(self, name):
        self.name = name
```

```
class Employment:
    def __init__(self, person, company):
        self.person = person
        self.company = company

# Асоціація між особою та компанією через зв'язок працевлаштування
john = Person("John")
acme = Company("ACME")
employment = Employment(john, acme)
```

2. Агрегація:

```
class Wheel:
    def __init__(self, model):
        self.model = model

class Car:
    def __init__(self, wheels):
        self.wheels = wheels

wheel1 = Wheel("Front left")
wheel2 = Wheel("Front right")
wheel3 = Wheel("Back left")
wheel4 = Wheel("Back right")

car = Car([wheel1, wheel2, wheel3, wheel4])  # Автомобіль має агрегацію коліс
```

3. Композиція:

```
class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower
```

```
class Car:
    def __init__(self, engine):
        self.engine = engine

engine = Engine(200)
car = Car(engine)  # Композиція автомобіля та двигуна
```

4. Асоціація:

```
class Library:
    def __init__(self, name):
        self.name = name

class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

class Membership:
    def __init__(self, person, library):
        self.person = person
        self.library = library

person1 = Person("Alice")
library1 = Library("Central Library")
membership = Membership(person1, library1)
```

5. Агрегація:

```
class University:
    def __init__(self, name):
        self.name = name

class Student:
    def __init__(self, name):
        self.name = name
```

```
class Department:
    def __init__(self, name, students):
        self.name = name
        self.students = students

student1 = Student("John")
student2 = Student("Alice")
student3 = Student("Bob")

computer_science_students = [student1, student2, student3]
computer_science_department = Department("Computer Science",
computer_science_students)
```

У цих прикладах демонструється використання різних типів взаємозв'язків між класами залежно від ступеня залежності та життєвого циклу об'єктів.

Клас у класі, відомий також як вкладений клас або вкладений тип, – це клас, який оголошений всередині іншого класу. Вкладені класи корисні для організації логічних груп об'єктів та зменшення простору імен, а також для створення спеціалізованих типів даних, які є властивими тільки для певного класу.

1. Клас як засіб організації:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.engine = self.Engine()  # Вкладений клас Engine

    class Engine:  # Вкладений клас
        def __init__(self, horsepower):
            self.horsepower = horsepower
```

```
# Створення об'єкта автомобіля
car = Car("Toyota", "Camry", 2022)
# Створення об'єкта двигуна
engine = car.engine
print(engine.horsepower)  # Вивід потужності двигуна
```

2. Клас як спеціалізований тип даних:

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    class Node:  # Вкладений клас
        def __init__(self, data):
            self.data = data
            self.next = None

# Створення об'єкта стеку
stack = Stack()
# Створення об'єкта вузла
node = stack.Node(10)
print(node.data)  # Вивід даних вузла
```

3. Клас як спосіб організації даних та методів:

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.marks = self.Marks()  # Вкладений клас Marks
```

```
    class Marks:  # Вкладений клас
        def __init__(self):
            self.subjects = {}

        def add_mark(self, subject, mark):
            self.subjects[subject] = mark

# Створення об'єкта студента
student = Student("Alice", 20)
# Додавання оцінок
student.marks.add_mark("Math", 90)
student.marks.add_mark("Physics", 85)
print(student.marks.subjects)  # Вивід оцінок
```

Ці приклади демонструють, як вкладені класи можуть бути використані для створення більш структурованого та модульного коду в Python. Вкладені класи забезпечують ієрархічну структуру даних та методів, що може бути корисною для організації складних систем або об’єктів.

21.7. Написання класів-домішок

Класи-домішки (*mixin classes*) – це класи, які містять функціональність, яка може бути використана в багатьох інших класах за допомогою механізму множинного успадкування. Це дає змогу перевикористовувати код із класів-домішок у різних класах без необхідності повторного написання коду. Приклади класів-домішок:

1. Логування:

```
class LogMixin:
    def log(self, message):
        print(f"[{self.__class__.__name__}] {message}")
```

```
class MyClass(LogMixin):
    def some_method(self):
        self.log("This is a log message")

obj = MyClass()
obj.some_method()
```

У цьому прикладі *LogMixin* є класом-домішкою, який містить метод *log()*. Клас *MyClass* успадковує цей метод, тому об'єкти класу *MyClass* можуть використовувати метод log() для запису логів.

2. Серіалізація в JSON:

```
import json

class JSONMixin:
    def to_json(self):
        return json.dumps(self.__dict__)

class Person(JSONMixin):
    def __init__(self, name, age):
        self.name = name
        self.age = age

person = Person("Alice", 30)
print(person.to_json())
```

У цьому прикладі *JSONMixin* містить метод *to_json()*, який перетворює атрибути об'єкта на рядок у форматі JSON. Клас *Person* успадковує цей метод, щоб об'єкти *Person* могли бути легко серіалізовані у JSON.

3. Конвертація в рядок:

```
class StringMixin:
    def __str__(self):
        return f"{self.__class__.__name__}({self.__dict__})"

class Point(StringMixin):
    def __init__(self, x, y):
        self.x = x
        self.y = y

point = Point(10, 20)
print(point)
```

У цьому прикладі *StringMixin* містить метод *__str__()*, який повертає рядок, представлення об'єкта у зручному для читання форматі. Клас *Poin*t успадковує цей метод, щоб об'єкти *Point* могли бути зручно виведені у вигляді рядка.

4. Кешування результатів обчислень:

```
class CacheMixin:
    _cache = {}

    def cached_method(self, key):
        if key not in self._cache:
            self._cache[key] = self.compute_value(key)
        return self._cache[key]

    def compute_value(self, key):
        # Метод, який обчислює значення за ключем
        pass

class Calculator(CacheMixin):
    def compute_value(self, key):
        return key * 2
```

```
calculator = Calculator()
print(calculator.cached_method(5))  # Обчислення результату та кешування
print(calculator.cached_method(5))  # Використання кешованого результату
```

У цьому прикладі *CacheMixin* містить метод *cached_method()*, який кешує результати обчислень для подальшого використання. Клас *Calculator* успадковує цей метод, а також метод *compute_value()*, який фактично обчислює значення. Отже, чином, клас *Calculator* може використовувати кеш для оптимізації обчислень.

Перелік питань для самоконтролю

1. Спадкування.
2. Поняття базового класу (суперкласу) та похідного класу (підкласу).
3. Множинне спадкування.
4. Асоціація.
5. Агрегація.
6. Композиція.
7. Використання функції super().
8. Класи-домішки.

Рекомендовані джерела:

Основні: 1; 2; 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Створіть базовий клас `Animal` з методом `make_sound()`, який виводить "Some sound". Створіть похідний клас `Dog`, який перевизначає метод `make_sound()` для виводу "Woof".

2. Створіть базовий клас Person з атрибутами name і age. Створіть похідний клас `Student`, який має додатковий атрибут `student_id` і переініціалізує конструктор базового класу.

3. Створіть базовий клас `Shape` з методом `area()`, який повертає 0. Створіть похідні класи `Rectangle` та `Circle`, які перевизначають метод `area()` для обчислення площі відповідних фігур.

4. Створіть клас `Employee` з методом класу `get_company_name()`. Створіть похідний клас `Manager`, який використовує метод класу `get_company_name()`.

5. Створіть базовий клас `Vehicle` з приватним атрибутом `__model`. Створіть похідний клас Car, який намагається отримати доступ до цього приватного атрибута.

6. Створіть два базових класи `Flying` і `Swimming` з методами `fly()` і `swim()` відповідно. Створіть похідний клас `Duck`, який успадковує від обох базових класів і реалізовує обидва методи.

7. Створіть базовий клас `Animal` з методом `describe()`. Створіть похідний клас `Cat`, який перевизначає метод `describe()` для додавання додаткового опису.

8. Створіть базовий клас `Rectangle` з властивостями `width` і `height`. Створіть похідний клас `Square`, який автоматично оновлює обидві властивості під час змінення одного з них.

9. Створіть абстрактний клас `Shape` з абстрактним методом `draw()`. Створіть похідний класи `Circle` і `Rectangle`, які реалізують метод `draw()`.

10. Створіть базовий клас `Vehicle` з атрибутом `type`. Створіть похідний клас `ElectricCar` і вирішіть проблему, коли конструктор базового класу не викликається належним чином.

11. Створіть клас-домішок `ColorMixin`, який має метод `set_color()` для встановлення кольору і метод `get_color()` для отримання кольору. Потім створіть основний клас `Rectangle`, який використовує `ColorMixin` для додавання можливості роботи з кольором.

12. Створіть клас-домішок `LoggerMixin`, який має метод `log()` для запису повідомлень у консоль. Створіть основний клас `Application`, який використовує `LoggerMixin` для додавання можливості логування подій.

13. Створіть клас-домішок `AreaMixin`, який має метод `calculate_area()` для обчислення площі. Потім створіть класи `Circle` і `Rectangle`, які використовують `AreaMixin` для надання можливості розрахунку площі.

14. Створіть клас-домішок `ValidationMixin`, який має метод `validate()` для перевірки валідності даних. Створіть основний клас `UserProfile`, який використовує `ValidationMixin` для перевірки коректності введених даних профілю.

15. Створіть клас-домішок `TimingMixin`, який має метод `log_time()` для підрахунку часу виконання функцій. Створіть основний клас `Task`, який використовує `TimingMixin` для вимірювання часу виконання методів.

16. Створіть два класи `Author` і `Book`. Клас `Book` має атрибут, що посилається на об’єкт класу `Author`. Клас `Author` може мати кілька книг.

17. Створіть клас `Category`, який містить список об’єктів класу `Product`. Клас `Product` не має посилання назад на клас `Category`. Це приклад агрегації, де категорія агрегує продукти, але продукти можуть існувати самостійно.

18. Створіть клас `Computer`, який має складання компонентів: `CPU`, `RAM` і `HardDrive`. Класи `CPU`, `RAM` і `HardDrive` є частинами класу `Computer`, і їх життєвий цикл залежить від об’єкта `Computer`.

19. Створіть клас Person, який має атрибут `address`, що є об'єктом класу `Address`. Клас `Address` не має посилання назад на Person. Це приклад асоціації.

20. Створіть клас `Client`, який містить список об’єктів класу `Order`. Клас `Order` не має посилання назад на `Client`. Клієнт може мати кілька замовлень, але замовлення можуть існувати без клієнта.

21. Створіть клас Car, який містить об’єкти класів `Engine`, `Wheel` і `Transmission`. Частини автомобіля (двигун, колеса, трансмісія) є частинами автомобіля і не можуть існувати без нього.

22. Створіть клас `Course`, який має список студентів (об’єктів класу `Student`). Клас `Student` не має посилання назад на `Course`. Це приклад асоціації.

23. Створіть клас `Company`, який містить список об’єктів класу `Project`. Клас `Project` не має посилання назад на `Company`. Компанія може мати кілька проєктів, але проєкти можуть існувати без компанії.

24. Створіть клас `Building`, який має список об’єктів класу `Room`. Кімнати є частинами будівлі і не можуть існувати без неї. Під час видалення об’єкта `Building` відповідні кімнати також повинні бути видалені.

25. Створіть клас `Album`, який має список об’єктів класу `Song`. Клас `Song` не має посилання назад на `Album`. Це приклад асоціації, де альбом має пісні, але пісні можуть існувати самостійно.

Тема 22
Спеціальні методи

22.1. Спеціальні методи класів у Python, загальні поняття.

22.2. Методи перевантаження операторів роботи з колекціями.

22.3. Основні методи для операцій над двійковими числами.

22.4. Комбіновані методи для операцій над двійковими числами.

22.5. Методи для операцій з дескрипторами.

22.6. Методи для операцій, що використовуються з диспетчерами контексту.

22.1. Спеціальні методи класів у Python, загальні поняття

Спеціальні методи в Python – це методи, які починаються і закінчуються подвійними підкресленнями, такі як `__init__`, `__str__`, `__repr__` тощо. Вони також відомі як «магічні методи» і надають класам можливість інтегруватися з основними механізмами Python, такими як оператори, функції, і стандартні методи об'єктів.

Основні спеціальні методи.

1. *__init__(self, ...)* -– конструктор класу. Викликається під час створення нового об'єкта. Ініціалізує атрибути об'єкта.

```
class MyClass:
    def __init__(self, value):
        self.value = value
```

2. *__str__(self)* – використовується функцією *str()* і функцією *print()* для отримання «зручного» рядкового представлення об'єкта.

```
class MyClass:
    def __str__(self):
        return f"MyClass with value {self.value}"
```

3. *__repr__(self)* – використовується функцією *repr()* для отримання «форматованого» рядкового представлення об’єкта, яке має бути корисним для налагодження і можливого відновлення об’єкта.

```
class MyClass:
    def __repr__(self):
        return f"MyClass({self.value!r})"
```

4. *__len__(self)* – викликається функцією *len()* для отримання довжини об’єкта.

```
class MyCollection:
    def __init__(self, items):
        self.items = items

    def __len__(self):
        return len(self.items)
```

5. *__getitem__(self, key)* – викликається для отримання елемента з колекції за індексом або ключем (як у словниках).

```
class MyCollection:
    def __getitem__(self, index):
        return self.items[index]
```

6. *__setitem__(self, key, value)* – викликається для встановлення значення в колекцію за індексом або ключем.

```
class MyCollection:
    def __setitem__(self, index, value):
        self.items[index] = value
```

7. *__delitem__(self, key)* – викликається для видалення елемента з колекції за індексом або ключем.

```
class MyCollection:
    def __delitem__(self, index):
        del self.items[index]
```

8. *__iter__(self)* – повертає ітератор для обходу об’єкта в циклі *for*.

```
class MyCollection:
    def __iter__(self):
        return iter(self.items)
```

9. *__next__(self)* – використовується ітератором для повернення наступного елемента під час ітерації.

```
class MyIterator:
    def __init__(self, items):
        self.items = items
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.items):
            raise StopIteration
        result = self.items[self.index]
        self.index += 1
        return result
```

10. *__contains__(self, item)* – викликається функцією *in* для перевірки наявності елемента в колекції.

```
class MyCollection:
    def __contains__(self, item):
        return item in self.items
```

11. *__eq__(self, other)* – використовується для порівняння об’єктів на рівність за допомогою оператора `==`.

```
class MyClass:
    def __eq__(self, other):
        return self.value == other.value
```

12. *__add__(self, other)* – викликається для додавання об'єктів за допомогою оператора `+`.

```
class MyClass:
    def __add__(self, other):
        return MyClass(self.value + other.value)
```

13. *__call__(self, ...)* – дає змогу об'єктам класу викликатися як функції.

```
class MyCallable:
    def __init__(self, value):
        self.value = value

    def __call__(self, increment):
        return self.value + increment
```

14. *__enter__(self)* і *__exit__(self, exc_type, exc_val, exc_tb)* – використовуються для реалізації контекстного менеджера з оператором *with*.

```
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")
        return False
```

Клас *Vector*, який використовує кілька спеціальних методів:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __len__(self):
        return 2  # Вектор завжди має два компоненти

    def __call__(self):
        return (self.x, self.y)

# Використання
v1 = Vector(1, 2)
v2 = Vector(3, 4)

print(v1)             # Виведе: Vector(1, 2)
print(v1 + v2)        # Виведе: Vector(4, 6)
print(v1 == Vector(1, 2)) # Виведе: True
print(len(v1))        # Виведе: 2
print(v1())           # Виведе: (1, 2)
```

Переваги використання спеціальних методів:

- інтеграція з синтаксисом Python – дає змогу вашим класам працювати з основними синтаксичними конструкціями і функціями Python;

- гнучкість і розширюваність – дає змогу легко реалізувати поведінку, схожу на вбудовані типи даних;

- полегшення коду – зменшує необхідність у прямому використанні методів і надає можливість використовувати природний синтаксис для операцій.

22.2. Методи перевантаження операторів роботи з колекціями

Методи перевантаження операторів роботи з колекціями дають змогу визначати спеціальну поведінку для об’єктів під час виконання різних операцій, таких як індексування, додавання, видалення тощо. Ось декілька методів перевантаження операторів, які можна використовувати з колекціями:

1. Метод *__getitem__()* для індексування:

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __getitem__(self, index):
        return self.items[index]

# Використання індексування для об'єкта MyList
my_list = MyList([1, 2, 3, 4, 5])
print(my_list[2])  # Виведе 3
```

2. Метод *__setitem__()* для зміни елементів за індексом:

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __setitem__(self, index, value):
        self.items[index] = value

# Зміна елемента за індексом в об'єкті MyList
my_list = MyList([1, 2, 3, 4, 5])
my_list[2] = 10
print(my_list)  # Виведе [1, 2, 10, 4, 5]
```

3. Метод *__delitem__()* для видалення елементів за індексом:

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __delitem__(self, index):
        del self.items[index]

# Видалення елемента за індексом з об'єкта MyList
my_list = MyList([1, 2, 3, 4, 5])
del my_list[2]
print(my_list)  # Виведе [1, 2, 4, 5]
```

4. Метод *__len__()* для визначення довжини колекції:

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __len__(self):
        return len(self.items)

# Визначення довжини об'єкта MyList
my_list = MyList([1, 2, 3, 4, 5])
print(len(my_list))  # Виведе 5
```

5. Метод *__contains__()* для перевірки наявності елемента в колекції:

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __contains__(self, item):
        return item in self.items
```

```
# Перевірка наявності елемента в об'єкті MyList
my_list = MyList([1, 2, 3, 4, 5])
print(3 in my_list)  # Виведе True
```

6. Метод `__iter__()` для ітерації по колекції:

```
class MyList:
    def __init__(self, items):
        self.items = items

    def __iter__(self):
        return iter(self.items)

# Ітерація по об'єкту MyList
my_list = MyList([1, 2, 3, 4, 5])
for item in my_list:
    print(item)  # Виведе числа від 1 до 5
```

22.3. Основні методи для операцій над двійковими числами

Для роботи з двійковими числами у Python можна використовувати вбудовані типи даних, такі як *int*, і використовувати різні методи цих класів для виконання різних операцій. Ось декілька основних методів класів для операцій над двійковими числами:

1. Перетворення у двійкову строку:

```
num = 10
binary_str = bin(num)
print(binary_str)  # Виведе '0b1010'
```

2. Перетворення з двійкової строки в ціле число:

```
binary_str = '1010'
num = int(binary_str, 2)
print(num)  # Виведе 10
```

3. Бітова кон’юнкція *(AND):*

```
num1 = 0b1010
num2 = 0b1100
result = num1.__and__(num2)
print(bin(result))  # Виведе '0b1000'
```

4. Бітова диз’юнкція *(OR):*

```
num1 = 0b1010
num2 = 0b1100
result = num1.__or__(num2)
print(bin(result))  # Виведе '0b1110'
```

5. Виключне *АБО (XOR):*

```
num1 = 0b1010
num2 = 0b1100
result = num1.__xor__(num2)
print(bin(result))  # Виведе '0b0110'
```

6. Зсув вправо:

```
num = 0b1010
result = num.__rshift__(1)
print(bin(result))  # Виведе '0b0101'
```

7. Зсув вліво:

```
num = 0b1010
result = num.__lshift__(1)
print(bin(result))  # Виведе '0b10100'
```

Ці методи надають можливість виконувати різні операції над двійковими числами за допомогою вбудованих методів класів `int` у Python. Вони є альтернативою використанню операторів бітових операцій (`&`, `|`, `^`, `>>`, `<<`).

22.4. Комбіновані методи для операцій над двійковими числами

Комбіновані методи для операцій над двійковими числами дають змогу виконувати операції зі змінною та присвоювати результат змінній в одному виразі. У Python для цього можна використовувати комбіновані оператори разом із бітовими операціями.

1. Комбінована бітова кон'юнкція *(AND):*

```
num1 = 0b1010
num2 = 0b1100
num1 &= num2
print(bin(num1))  # Виведе '0b1000'
```

2. Комбінована бітова диз'юнкція *(OR):*

```
num1 = 0b1010
num2 = 0b1100
num1 |= num2
print(bin(num1))  # Виведе '0b1110'
```

3. Комбіноване виключне *АБО (XOR):*

```
num1 = 0b1010
num2 = 0b1100
num1 ^= num2
print(bin(num1))  # Виведе '0b0110'
```

4. Комбінований зсув вправо:

```
num = 0b1010
num >>= 1
print(bin(num))  # Виведе '0b0101'
```

5. Комбінований зсув вліво:

```
num = 0b1010
num <<= 1
print(bin(num))  # Виведе '0b10100'
```

Ці комбіновані методи дають змогу виконувати бітові операції та змінювати значення змінних у одному виразі. Вони є коротшими та більш ефективними порівняно з використанням комбінованих виразів типу *num = num1 & num2*.

22.5. Методи для операцій з дескрипторами

Методи для операцій з дескрипторами в Python дають змогу визначити поведінку атрибутів, коли вони доступні через дескриптори. Дескриптори є об'єктами, які визначають, як класи мають поводитися під час доступу до їх атрибутів. Ось декілька основних методів, які можна використовувати для операцій з дескрипторами:

1. *__get__(self, instance, owner)* – цей метод викликається, коли здійснюється доступ до атрибута через дескриптор. Параметр *self* – сам дескриптор, *instance* – екземпляр класу, до якого відноситься атрибут, а *owner* – клас, до якого належить атрибут. Цей метод повинен повернути значення атрибута.

2. *__set__(self, instance, value)* – викликається, коли відбувається присвоєння значення атрибуту через дескриптор. Параметр *self* – сам дескриптор, *instance* – екземпляр класу, до якого відноситься атрибут, а *value* – значення, яке призначається атрибуту. Цей метод може виконати додаткову обробку перед присвоєнням значення атрибуту.

3. *__delete__(self, instance)* – викликається під час видалення атрибута через дескриптор. Параметр *self* – сам дескриптор, а *instance* – екземпляр класу, до якого відноситься атрибут. Цей метод може виконати додаткову обробку перед видаленням атрибута.

```
class Descriptor:
    def __get__(self, instance, owner):
        print("Getting the attribute")
        return instance._value

    def __set__(self, instance, value):
        print("Setting the attribute")
        instance._value = value * 2

    def __delete__(self, instance):
        print("Deleting the attribute")
        del instance._value

class MyClass:
    descriptor = Descriptor()

    def __init__(self, value):
        self._value = value

obj = MyClass(10)
print(obj.descriptor)  # Виведе "Getting the attribute" і 10
obj.descriptor = 5
print(obj.descriptor)  # Виведе "Setting the attribute",
# а потім "Getting the attribute" і 10
del obj.descriptor
```

У цьому прикладі *Descriptor* – це дескриптор, який додає до атрибута *descriptor* класу *MyClass* спеціальну поведінку під час доступу до нього. Методи *__get__()*, *__set__()* і *__delete__()* викликаються відповідно під час зчитування, присвоєння та видалення значення атрибута.

22.6. Методи для операцій, що використовуються з диспетчерами контексту

Методи, що використовуються з диспетчерами контексту (менеджерами контексту) в Python, дають змогу визначати поведінку об'єктів, які використовуються в контексті оператора *with*.

1. *__enter__(self)* – цей метод викликається під час входу в блок контексту (після ключового слова *with*). Він повинен повернути об'єкт, який буде прив'язаний до змінної, яка зазначена після *as*.

2. *__exit__(self, exc_type, exc_value, traceback)* – викликається під час виходу з блоку контексту, навіть якщо відбулася помилка в блоці. Параметри *exc_type*, *exc_value* і *traceback* містять інформацію про будь-яку виняткову ситуацію, яка виникла в блоці контексту. Якщо виняток не було піднято, всі три параметри мають значення *None*. Якщо метод *__exit__* повертає *True*, то будь-який виняток, який було піднято в блоці контексту, буде поглинутий і програма продовжить виконання після блоку контексту. Якщо метод *__exit__* повертає *False* або *None*, то будь-який виняток буде піднятий знову після виклику методу *__exit__*.

```
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")
        if exc_type:
            print(f"Exception occurred: {exc_type}, {exc_value}")

# Використання диспетчера контексту
with MyContextManager() as cm:
    print("Inside the context")
    # Відобразить "Inside the context"
    # Потім "Exiting the context"
```

У цьому прикладі метод *__enter__* просто повертає об’єкт диспетчера контексту, який дозволяє доступ до нього після ключового слова *as* у виразі *with*. Під час виходу з контексту метод *__exit__* виводить повідомлення про вихід з контексту. Якщо виникла виняткова ситуація, вона також буде виведена, але вона буде оброблена і програма продовжить виконання.

Перелік питань для самоконтролю

1. Звертання до класів інших модулів. Загальноприйняті домовленості до задавання імен модулів та класів.
2. Методи для всіх видів операцій.
3. Методи перевантаження операторів роботи з колекціями.
4. Основні методи для операцій над двійковими числами.
5. Методи для правосторонніх операцій над двійковими числами.
6. Комбіновані методи для операцій над двійковими числами.
7. Методи для інших операцій над числами.
8. Методи для операцій з дескрипторами.
9. Методи для операцій, що використовуються з диспетчерами контексту.

Тестові питання

1. Який спеціальний метод класу викликається під час створення нового об’єкта класу:

а) `__str__`;
б) `__init__`;
в) `__repr__`;
г) `__call__`.

2. Який спеціальний метод класу використовується для отримання рядкового представлення об'єкта, яке зручно для читання:

а) `__repr__`;
б) `__str__`;
в) `__format__`;
г) `__eq__`.

3. Який спеціальний метод класу дає змогу використовувати оператор `+` для об'єктів класу:

а) `__mul__`;
б) `__sub__`;
в) `__add__`;
г) `__pow__`.

4. Який спеціальний метод класу викликається для перевірки на рівність двох об'єктів за допомогою оператора `==`:

а) `__gt__`;
б) `__eq__`;
в) `__ne__`;
г) `__lt__`.

5. Який спеціальний метод класу використовується для доступу до елемента в колекції за індексом або ключем:

а) `__getitem__`;
б) `__setitem__`;
в) `__delitem__`;
г) `__contains__`.

6. Який спеціальний метод класу викликається для видалення елемента з колекції за індексом або ключем:

а) `__del__`;
б) `__delitem__`;
в) `__remove__`;
г) `__discard__`.

7. Який спеціальний метод класу дає змогу об’єктам класу бути ітерабельними в циклі `for`:

а) `__next__`;
б) `__iter__`;
в) `__contains__`;
г) `__call__`.

8. Який спеціальний метод класу дає змогу об’єктам класу використовуватися як функції (тобто бути викликаними як функції):

а) `__call__`;
б) `__apply__`;
в) `__exec__`;
г) `__invoke__`.

9. Який спеціальний метод класу викликається, коли об’єкт класу входить у контекст `with` (управляється контекстним менеджером):

а) `__enter__`;
б) `__start__`;
в) `__open__`;
г) `__context__`.

10. Який спеціальний метод класу викликається під час порівняння об’єктів з використанням оператора `<`:

а) `__le__`;
б) `__lt__`;
в) `__ge__`;
г) `__gt__`.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Створіть клас `Book`, який має атрибути `title`, `author` та `year`. Реалізуйте конструктор `__init__` для ініціалізації цих атрибутів і метод `__str__` для зручного представлення інформації про книгу у вигляді рядка, наприклад, `"Title by Author (Year)"`.

2. Розширте попередній клас `Book` і додайте метод `__repr__`, який повертає рядок у форматі `Book(title='Title', author='Author', year=Year)`, що дає змогу відновити об’єкт за допомогою `eval`.

3. Створіть клас `Team`, який має атрибут `members`, що є списком імен учасників. Реалізуйте метод `__len__`, який повертає кількість учасників у команді.

4. Створіть клас `Matrix`, який має атрибут `data`, що є списком списків. Реалізуйте метод `__getitem__` для доступу до елементів матриці за допомогою індексів, наприклад, `matrix[1][2]`.

5. Розширте клас `Matrix` і додайте метод `__setitem__` для зміни значень у матриці, наприклад, `matrix[1][2] = 5`.

6. Створіть клас `CustomList`, який має атрибут `items`, що є списком. Реалізуйте метод `__delitem__` для видалення елементів з цього списку, наприклад, `del custom_list[2]`.

7. Розширте клас `CustomList` і реалізуйте метод `__iter__`, щоб можна було ітеруватися за списком за допомогою циклу `for`.

8. Створіть клас `Multiplier`, який має атрибут `factor`. Реалізуйте метод `__call__`, що дає змогу об’єкту цього класу множити передане число на значення `factor`. Наприклад, `multiplier(4)` поверне `factor * 4`.

9. Створіть клас `Circle`, який має атрибут `radius`. Реалізуйте метод `__eq__`, щоб порівнювати два об’єкти `Circle` на основі їхнього радіуса. Два кола повинні вважатися рівними, якщо їхні радіуси однакові.

10. Створіть клас `FileWriter`, який відкриває файл для запису. Реалізуйте методи `__enter__` і `__exit__`, щоб можна було використовувати цей клас у конструкції `with`. Відкривайте файл у методі `__enter__` і закривайте його в методі `__exit__`.

11. Створіть клас `Vector`, що представляє вектор у двовимірному просторі. Перезавантажте оператор `+`, щоб дозволити додавання двох векторів.

12. Розширте клас `Vector`, щоб включити перезавантаження оператора `-` для віднімання двох векторів.

13. Розширте клас `Vector`, щоб включити перезавантаження оператора `*`, який множить вектор на скаляр.

14. Створіть клас `Rectangle`, який представляє прямокутник з шириною і висотою. Перезавантажте оператор `==`, щоб перевірити, чи два прямокутники мають однакову площу.

15. Розширте клас `Rectangle`, щоб включити перезавантаження оператора `<`, що порівнює площі двох прямокутників.

16. Створіть клас `Matrix`, який представляє матрицю. Перезавантажте оператор `[]`, щоб дозволити доступ до елементів матриці за допомогою подвійних квадратних дужок, наприклад, `matrix[i][j]`.

17. Створіть клас `Multiplier`, який використовує метод `__call__`, щоб дозволити об’єкту цього класу множити значення, яке передається як аргумент.

18. Розширте клас `Person`, щоб включити методи `__str__` і `__repr__`, які забезпечують зрозуміле та детальне представлення об’єктів.

19. Створіть клас `Inventory`, який представляє список предметів. Перезавантажте оператор `in`, щоб перевіряти наявність предметів у інвентарі.

20. Створіть клас `FileLogger`, який відкриває файл для запису. Реалізуйте методи `__enter__` і `__exit__` для використання класу як контекстного менеджера.

Тема 23
Поліморфізм та перезавантаження спеціальних методів

23.1. Поліморфізм.

23.2. Загальні відомості про перевантаження операторів. Принципи, що лежать в основі перевантаження операторів у класах.

23.3. Перелік методів, які можна перевантажувати.

23.4. Перевантаження доступу за індексом []. Метод __getitem__().

23.5. Перевантаження доступу за індексом [ ]. Встановлення нового значення. Метод __setitem__().

23.6. Віртуальні методи.

23.7. Метод __getattribute__(), який викликається під час доступу до будь-якого атрибута класу.

23.8. Метод __setattr__(), який викликається під час спроби присвоювання значення атрибуту екземпляра класу.

23.9. Метод __delattr__(), який викликається під час видалення атрибута за допомогою інструкції del.

23.1. Поліморфізм

Поліморфізм – це можливість об’єктів з одного класу вести себе по-різному залежно від контексту, в якому вони використовуються. У Python поліморфізм може бути досягнутий за допомогою унаслідування, використання абстрактних класів або просто через використання об’єктів, які мають спільний інтерфейс.

1. Поліморфізм за допомогою унаслідування:

```
class Animal:
    def speak(self):
        pass
```

```
class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Використання поліморфізму
animals = [Dog(), Cat()]
for animal in animals:
    print(animal.speak())  # Викликає метод speak() для кожного об'єкта, але результат відрізняється залежно від типу об'єкта
```

У цьому прикладі *Dog* та *Cat* є підкласами класу *Animal*. Кожен з цих підкласів має свою власну реалізацію методу *speak()*, що представляє різні звуки, які вони видають. Під час виклику методу *speak()* для кожного об’єкта повертається різний звук.

2. Поліморфізм за допомогою абстрактних класів:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius  2

# Використання поліморфізму
shapes = [Rectangle(4, 5), Circle(3)]
for shape in shapes:
    print(shape.area())  # Викликає метод area() для кожного об'єкта, але результат відрізняється залежно від типу об'єкта
```

У цьому прикладі *Rectangle* та *Circle* є підкласами абстрактного класу *Shape*. Кожен з цих підкласів реалізує метод *area()*, але робить це по-різному для розрахунку площі. Під час виклику методу *area()* для кожного об’єкта отримуємо різні результати.

3. Поліморфізм за допомогою спільного інтерфейсу:

```
class Printer:
    def print(self, text):
        pass

class ConsolePrinter(Printer):
    def print(self, text):
        print("Printing to console:", text)

class FilePrinter(Printer):
    def print(self, text):
        with open("output.txt", "w") as f:
            f.write(text)
```

```
# Використання поліморфізму
printers = [ConsolePrinter(), FilePrinter()]
for printer in printers:
    printer.print("Hello, world!")  # Викликає метод print() для кожного об'єкта, але результат відрізняється залежно від типу об'єкта
```

У цьому прикладі обидва класи *ConsolePrinter* та *FilePrinter* мають спільний інтерфейс, що представлений методом *print()*. Кожен з цих класів виконує друк тексту, але робить це по-різному: один виводить текст у консоль, а інший записує його у файл. Під час виклику методу *print()* для кожного об’єкта отримуємо різні результати відповідно до його типу.

23.2. Загальні відомості про перевантаження операторів. Принципи, що лежать в основі перевантаження операторів у класах

Перевантаження операторів (*operator overloading*) – це механізм у Python, який дає змогу змінювати поведінку вбудованих операторів для власних класів. Завдяки перевантаженню операторів можна забезпечити спеціальну поведінку об’єктів класу під час використання операторів мови.

1. Оператори, що можуть бути перевантажені. Більшість операторів у Python можуть бути перевантажені для користувальницьких класів. Це охоплює арифметичні оператори (+, -, *, /), порівняння (<, >, ==), логічні оператори (and, or, not), індексацію ([]) та інші.

2. Магічні методи (спеціальні методи). Перевантаження операторів у Python здійснюється за допомогою спеціальних методів класу, які називаються магічними методами або методами перевантаження операторів. Ці методи мають певний синтаксис і викликаються автоматично під час використання відповідного оператора.

3. Приклади магічних методів. Наприклад, для перевантаження оператора `+` потрібно визначити метод `__add__`, для `*` – `__mul__`, для порівняння – `__lt__`, `__eq__` та інші. Повний список магічних методів можна знайти в документації Python.

4. Застосування перевантаження операторів. Перевантаження операторів може бути корисним у випадках, коли ви хочете надати класу спеціальну поведінку під час використання операторів. Наприклад, для коректного відображення об'єктів вашого класу у виразах арифметики або порівняння.

5. Переваги та недоліки. Перевантаження операторів може зробити ваш код більш зрозумілим та елегантним, забезпечуючи інтуїтивно зрозумілу поведінку. Однак варто користуватися цим з уважністю, оскільки надмірне використання перевантаження операторів може зробити код складним для розуміння.

Правильне використання перевантаження операторів може значно полегшити роботу з користувальницькими класами у Python, роблячи код більш зрозумілим і зручним для використання.

Перевантаження операторів у класах ґрунтується на кількох принципах, які допомагають забезпечити консистентність та передбачуваність поведінки об'єктів під час використання вбудованих операторів мови. Основні принципи перевантаження операторів охоплюють:

1. Магічні методи. Перевантаження операторів у Python відбувається за допомогою спеціальних методів класу, які називаються магічними методами або методами перевантаження операторів. Ці методи мають певний префікс та суфікс у назві, що вказує на оператор, який вони перевантажують. Наприклад, для оператора додавання (`+`) використовується метод *__add__*, для оператора порівняння (`<`) – *__lt__*, для індексації (`[]`) – *__getitem__* тощо.

2. Консистентність інтерфейсу. Перевантаження операторів повинно бути здійснене так, щоб забезпечити консистентність інтерфейсу. Це означає, що перевизначені оператори мають поводитися аналогічно вбудованим операторам, якщо це можливо. Наприклад, якщо ви перевизначаєте оператор додавання для свого класу, він повинен додавати об'єкти вашого класу так само, як вбудований оператор додавання додає числа.

3. Комутативність та асоціативність. Деякі оператори можуть мати комутативну (наприклад, `+`) або асоціативну (наприклад, `*`) властивості. У разі перевантаження цих операторів важливо забезпечити відповідність цим властивостям. Наприклад, якщо `A + B` рівноцінне `B + A`, то *__add__* повинен бути реалізований в обох напрямках.

4. Загальні конвенції. Під час перевантаження операторів важливо керуватися загальними конвенціями та очікуваннями спільноти Python. Наприклад, перевантаження оператора додавання (`+`) зазвичай використовується для об'єднання об'єктів, а перевантаження оператора порівняння (`<`) використовується для визначення відношення між об'єктами.

5. Документація. Перевантажені оператори повинні бути добре задокументовані, щоб інші розробники зрозуміли, яку поведінку очікувати від цих операторів під час використання вашого класу.

Дотримуючись цих принципів, ви забезпечите правильну та передбачувану поведінку вашого класу під час використання вбудованих операторів Python.

23.3. Перелік методів, які можна перевантажувати

У Python можна перевантажувати різноманітні оператори та вбудовані функції за допомогою спеціальних методів класу, які називаються магічними методами або методами перевантаження. Ось перелік деяких з найбільш поширених методів, які можна перевантажувати в Python:

1. Арифметичні оператори:
   - *`__add__(self, other)`: +*
   - *`__sub__(self, other)`: -*
   - *`__mul__(self, other)`: **
   - *`__truediv__(self, other)`: /*
   - *`__floordiv__(self, other)`: //*
   - *`__mod__(self, other)`: %*
   - *`__pow__(self, other[, modulo])`:*
   - *`__neg__(self)`: унарний -*

2. Оператори порівняння:
   - `__eq__(self, other)`: ==
   - `__ne__(self, other)`: !=
   - `__lt__(self, other)`: <
   - `__le__(self, other)`: <=
   - `__gt__(self, other)`: >
   - `__ge__(self, other)`: >=
3. Логічні оператори:
   - `__and__(self, other)`: and
   - `__or__(self, other)`: or
   - `__xor__(self, other)`: xor
   - `__not__(self)`: not
4. Оператори присвоєння:
   - `__iadd__(self, other)`: +=
   - `__isub__(self, other)`: -=
   - `__imul__(self, other)`: *=
   - `__itruediv__(self, other)`: /=
   - `__ifloordiv__(self, other)`: //=
   - `__imod__(self, other)`: %=
   - `__ipow__(self, other[, modulo])`: =
   - інші аналогічні оператори присвоєння
5. Інші:
   - `__str__(self)`: str()
   - `__repr__(self)`: repr()
   - `__len__(self)`: len()
   - `__getitem__(self, key)`: індексація зверху ([])
   - `__setitem__(self, key, value)`: присвоєння зверху ([])
   - `__iter__(self)`: ітерація
   - `__next__(self)`: наступний елемент ітерації
   - `__contains__(self, item)`: оператор *in*
   - `__enter__(self)`: контекстний менеджер *with*
   - `__exit__(self, exc_type, exc_value, traceback)`: контекстний менеджер *with*

Це лише декілька прикладів методів, які можна перевантажувати в Python. Кожен з цих методів дає змогу змінити поведінку класу під час використання вбудованих операторів чи функцій мови.

23.4. Перевантаження доступу за індексом []. Метод __getitem__()

Перевантаження доступу за індексом `[]` у Python можливе завдяки методу *__getitem__()*. Цей метод дає змогу класу створювати об'єкти, до яких можна звертатися за допомогою індексів, подібно до списків, словників або інших ітерабельних об'єктів.

```
class MyList:
    def __init__(self, *args):
        self.data = list(args)

    def __getitem__(self, index):
        return self.data[index]

# Приклад використання
my_list = MyList(1, 2, 3, 4, 5)
print(my_list[0])  # Виведе 1
print(my_list[2])  # Виведе 3
```

У цьому прикладі клас *MyList* перевизначає метод *__getitem__()*, який повертає елемент списку *data* за вказаним індексом. Тепер об'єкти класу *MyList* можуть бути індексовані як звичайні списки, і метод *__getitem__()* буде викликаний ппід час звертання до об'єкта за допомогою оператора `[]`.

Метод *__getitem__()* також може бути використаний для реалізації індексації в зворотному порядку, зрізів (*slicing*) та іншої логіки, пов'язаної з доступом за індексом до об'єкта.

```
class MyList:
    def __init__(self, *args):
        self.data = list(args)
```

```
    def __getitem__(self, index):
        return self.data[index]

    def __len__(self):
        return len(self.data)

# Використання зрізу (slicing)
my_list = MyList(1, 2, 3, 4, 5)
print(my_list[1:4])  # Виведе [2, 3, 4]

# Використання зворотного індекса
print(my_list[-1])  # Виведе 5

# Визначення довжини об'єкта
print(len(my_list))  # Виведе 5
```

Це дає змогу забезпечити зручний доступ до даних у ваших класах та надати їм схожого зі списками чи словниками інтерфейсу.

23.5. Перевантаження доступу за індексом [ ]. Встановлення нового значення. Метод __setitem__()

Метод *__setitem__()* використовується для перевантаження операції присвоєння за індексом `[]` у Python. Цей метод дає змогу класу встановлювати нове значення для певного елемента об’єкта, коли до нього звертаються за допомогою індексації.

```
class MyList:
    def __init__(self, *args):
        self.data = list(args)

    def __getitem__(self, index):
        return self.data[index]
```

```
    def __setitem__(self, index, value):
        self.data[index] = value

# Приклад використання
my_list = MyList(1, 2, 3, 4, 5)
print(my_list[2])  # Виведе 3
my_list[2] = 10
print(my_list[2])  # Тепер виведе 10
```

У цьому прикладі клас *MyLis*t визначає метод *__setitem__()*, який дає змогу встановлювати нове значення для елементів об'єкта *data* за вказаним індексом. Тепер об'єкти класу *MyList* можуть бути змінювані так само, як і списки, за допомогою оператора присвоєння за індексом `[]`.

Під час виклику *my_list[2] = 10* спершу викликається метод *__setitem__*, який установлює нове значення для елемента з індексом 2 у списку *data*, змінюючи його на 10. Після цього під час виклику *my_list[2]* повертається уже змінене значення 10.

23.6. Віртуальні методи

У Python віртуальні методи – це методи, що мають бути перевизначені в похідних класах. Їхній основний вигляд та поведінка визначаються в базовому класі, але вони мають бути перевизначені в дочірньому класі залежно від конкретної реалізації.

1. Абстрактний метод *draw* у класі *Shape*:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def draw(self):
        pass
```

```
class Circle(Shape):
    def draw(self):
        print("Drawing a circle")

class Square(Shape):
    def draw(self):
        print("Drawing a square")

circle = Circle()
square = Square()
circle.draw()  # Output: Drawing a circle
square.draw()  # Output: Drawing a square
```

2. Віртуальний метод *compute_area* у класі *Shape*:

```
class Shape:
    def compute_area(self):
        raise NotImplementedError("Subclasses must implement compute_area method")

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def compute_area(self):
        return 3.14 * self.radius  2

circle = Circle(5)
print(circle.compute_area())  # Output: 78.5
```

3. Віртуальний метод *run* у класі *Animal*:

```
class Animal:
    def run(self):
        raise NotImplementedError("Subclasses must implement run method")

class Dog(Animal):
    def run(self):
        print("Dog is running")

class Cat(Animal):
    def run(self):
        print("Cat is running")

dog = Dog()
cat = Cat()
dog.run()  # Output: Dog is running
cat.run()  # Output: Cat is running
```

4. Віртуальний метод *validate* у класі *Validator*:

```
class Validator:
    def validate(self, value):
        raise NotImplementedError("Subclasses must implement validate method")

class EmailValidator(Validator):
    def validate(self, email):
        return "@" in email

validator = EmailValidator()
print(validator.validate("example@example.com"))  # Output: True
```

5. Віртуальний метод *play* у класі *Game*:

```
class Game:
    def play(self):
        raise NotImplementedError("Subclasses must implement play method")

class TicTacToe(Game):
    def play(self):
        print("Playing Tic Tac Toe")

class Chess(Game):
    def play(self):
        print("Playing Chess")

tic_tac_toe = TicTacToe()
chess = Chess()
tic_tac_toe.play()  # Output: Playing Tic Tac Toe
chess.play()        # Output: Playing Chess
```

У цих прикладах методи визначаються в базовому класі з викиданням помилки *NotImplementedError*. Це змушує дочірні класи перевизначити ці методи.

23.7. Метод __getattribute__(), який викликається під час доступу до будь-якого атрибута класу

Метод *__getattribute__* викликається за спроби отримати доступ до будь-якого атрибута об’єкта класу в Python. Цей метод дає змогу Вам виконувати додаткові дії під час доступу до атрибутів. Однак важливо використовувати його з обережністю, оскільки неправильна реалізація може призвести до зациклювань або непередбачуваних результатів.

```
class Example:
    def __init__(self, value):
        self.value = value

    def __getattribute__(self, name):
        print(f"Accessing attribute: {name}")
        # Отримуємо значення атрибута через базовий метод
        return super().__getattribute__(name)

    def double_value(self):
        return self.value * 2

# Приклад використання
example = Example(42)
print(example.value)      # Виведе: Accessing attribute: value \n 42
print(example.double_value())    # Виведе: Accessing attribute: double_value \n 84
```

У цьому прикладі метод *__getattribute__* використовується для виведення повідомлення під час доступу до будь-якого атрибута об'єкта. Важливо пам'ятати, що виклик методу *__getattribute__* під час кожного доступу до атрибута може призвести до значного збільшення обчислювальних витрат, тому його використання потребує обережності та має бути обмеженим до необхідних ситуацій.

23.8. Метод __setattr__(), який викликається під час спроби присвоювання значення атрибуту екземпляра класу

Метод `*__setattr__*` викликається під час спроби присвоювання значення атрибуту екземпляра класу в Python. Цей метод дає змогу Вам виконувати додаткові дії перед тим, як значення буде присвоєно атрибуту.

```
class Example:
    def __init__(self):
        self._value = 0

    def __setattr__(self, name, value):
        print(f"Setting attribute: {name} = {value}")
        # Використовуємо базовий метод для присвоєння значення атрибуту
        super().__setattr__(name, value)

# Приклад використання
example = Example()
example.some_attribute = 42  # Виведе: Setting attribute: some_attribute = 42
print(example.some_attribute)  # Виведе: 42
```

У цьому прикладі метод *__setattr__* використовується для виведення повідомлення під час спроби присвоювання значення атрибуту. Після виведення повідомлення метод викликає базовий метод *super().__setattr__*, який фактично встановлює значення атрибуту.

Важливо пам’ятати, що виклик методу *__setattr__* під час кожного присвоєння атрибуту може призвести до значного збільшення обчислювальних витрат, тому його використання потребує обережності та має бути обмеженим до необхідних ситуацій.

23.9. Метод __delattr__(), який викликається під час видалення атрибута за допомогою інструкції del

Метод *__delattr__* викликається під час видалення атрибута за допомогою інструкції *del* у Python. Цей метод дає змогу виконувати додаткові дії перед тим, як атрибут буде видалено з екземпляра класу.

```
class Example:
    def __init__(self):
        self.some_attribute = 42

    def __delattr__(self, name):
        print(f"Deleting attribute: {name}")
        # Використовуємо базовий метод для видалення атрибута
        super().__delattr__(name)

# Приклад використання
example = Example()
print(example.some_attribute)  # Виведе: 42
del example.some_attribute          # Виведе: Deleting attribute: some_attribute
```

У цьому прикладі метод *__delattr__* використовується для виведення повідомлення перед видаленням атрибута. Після виведення повідомлення метод викликає базовий метод *super().__delattr__*, який фактично видаляє атрибут.

Важливо пам’ятати, що виклик методу *__delattr__* під час видалення атрибута може призвести до значного збільшення обчислювальних витрат, тому його використання потребує обережності та має бути обмеженим до необхідних ситуацій.

Перелік питань для самоконтролю

1. Поліморфізм.
2. Принципи, що лежать в основі перевантаження операторів у класах.
3. Перелік методів, які можна перевантажувати.
4. Перевантаження доступу за індексом. Метод __getitem__().
5. Перевантаження доступу за індексом. Встановлення нового значення. Метод __setitem__().
6. Перевантаження бінарних арифметичних операторів +, –, *, /, //, %.

7. Віртуальні методи.

8. Метод __getattribute__(), який викликається під час доступу до будь-якого атрибута класу.

9. Метод __setattr__(), який викликається під час спроби присвоювання значення атрибуту екземпляра класу.

10. Метод __delattr__(), який викликається під час видалення атрибута за допомогою інструкції del .

Тестові питання

1. Що таке поліморфізм у контексті об'єктно-орієнтованого програмування в Python:

а) можливість об'єктів одного типу бути використаними як об'єкти іншого типу;

б) можливість класу мати кілька конструкторів;

в) можливість методу мати різні ім'я в різних класах;

г) можливість методу змінювати свою поведінку залежно від типу об'єкта.

2. Який з наведених прикладів демонструє поліморфізм у Python:

а) `class Animal: def make_sound(self): pass`;

б) `def greet(person): print("Hello, " + person.name)`;

в) `class Dog(Animal): def make_sound(self): return "Bark"`;

г) `def add(x, y): return x + y`.

3. Як можна реалізувати поліморфізм під час використання функцій у Python:

а) використовуючи функції з однаковими іменами в різних модулях;

б) використовуючи функції з однаковими іменами, але з різними кількістю або типами параметрів;

в) використовуючи функції з однаковими іменами, але в різних класах;

г) використовуючи різні імена для кожної функції.

4. Яка функція використовується для перевірки типу об'єкта в Python:

а) `isinstance()`;

б) `type()`;

в) `checktype()`;

г) `kindof()`.

5. Який спеціальний метод дає змогує класу реалізувати поліморфізм для арифметичних операцій:

а) `__call__`;

б) `__str__`;

в) `__add__`;

г) `__getitem__`.

6. Які методи використовуються для реалізації поліморфізму в порівнянні об'єктів з використанням операторів:

а) `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`;

б) `__getitem__`, `__setitem__`, `__delitem__`;

в) `__iter__`, `__next__`;

г) `__str__`, `__repr__`.

7. Який метод потрібно реалізувати для забезпечення поліморфізму для оператора `*`:

а) `__add__`;

б) `__mul__`;

в) `__sub__`;

г) `__pow__`.

8. Яка особливість поліморфізму в Python дає змогу об'єктам одного типу використовуватися з різними типами даних:

а) інтерфейси;

б) типи даних;

в) спеціальні методи;

г) абстрактні класи.

9. Що таке «двaєдність» (overloading) у контексті поліморфізму в Python:

а) можливість методу мати декілька реалізацій з різними іменами;

б) можливість методу мати декілька реалізацій з однаковими іменами та різними параметрами;

в) можливість методу бути перевантаженим з різними типами результатів;

г) можливість методу бути викликаним з різними типами об’єктів.

10. Як реалізувати поліморфізм у Python під час роботи з ітераторами:

а) визначити метод `__iter__` у класі для повернення ітератора;

б) визначити метод `__next__` для об’єкта, який є ітератором;

в) визначити метод `__contains__` для перевірки наявності елемента;

г) визначити метод `__getitem__` для доступу до елементів.

Рекомендовані джерела:

Основні: 1; 2; 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Створіть базовий клас `Shape` з методом `area()`. Створіть підкласи `Square`, `Triangle` і `Ellipse`, які реалізують метод `area()` для обчислення площі відповідних фігур. Перевірте, як метод `area()` працює для різних типів фігур.

2. Створіть базовий клас `Formatter` з методом `format()`. Створіть підкласи `UpperCaseFormatter` і `LowerCaseFormatter`, які реалізують метод `format()` для перетворення рядка у верхній

або нижній регістр відповідно. Перевірте, як метод `format()` працює для різних форматувальників.

3. Створіть базовий клас `Statistics` з методом `mean()`. Створіть підкласи `MeanCalculator` і `MedianCalculator`, які реалізують метод `mean()` для обчислення середнього значення і медіани відповідно. Перевірте, як метод `mean()` працює для різних типів обчислень.

4. Створіть базовий клас `EventHandler` з методом `handle()`. Створіть підкласи `ClickEventHandler` і `KeyPressEventHandler`, які реалізують метод `handle()` для обробки подій натискання кнопок миші й клавіатури відповідно. Перевірте, як метод `handle()` обробляє різні події.

5. Створіть базовий клас `Employee` з методом `calculate_income()`. Створіть підкласи `SalariedEmployee` і `HourlyEmployee`, які реалізують метод `calculate_income()` для обчислення доходу за фіксованою зарплатою і погодинною оплатою відповідно. Перевірте, як метод `calculate_income()` працює для різних типів співробітників.

6. Створіть базовий клас `Message` з методом `send()`. Створіть підкласи `EmailMessage` і `SMSMessage`, які реалізують метод `send()` для надсилання електронних листів і SMS повідомлень відповідно. Перевірте, як метод `send()` працює для різних типів повідомлень.

7. Створіть базовий клас `Vehicle` з методом `start_engine()`. Створіть підкласи `Car` і `Motorcycle`, які реалізують метод `start_engine()` для запуску двигуна автомобіля і мотоцикла відповідно. Перевірте, як метод `start_engine()` працює для різних типів транспортних засобів.

8. Створіть базовий клас `Storage` з методом `store()`. Створіть підкласи `FileStorage` і `DatabaseStorage`, які реалізують метод `store()` для зберігання даних у файлі й базі даних відповідно. Перевірте, як метод `store()` працює для різних типів сховищ.

9. Створіть базовий клас `GraphicalElement` з методом `draw()`. Створіть підкласи `Circle` і `Rectangle`, які реалізують метод `draw()` для малювання кола і прямокутника відповідно. Перевірте, як метод `draw()` працює для різних графічних елементів.

10. Створіть базовий клас `FileProcessor` з методом `process()`. Створіть підкласи `TextFileProcessor` і `BinaryFileProcessor`, які реалізують метод `process()` для обробки текстових і бінарних файлів відповідно. Перевірте, як метод `process()` працює для різних типів файлів.

11. Створіть базовий клас `Report` з методом `generate()`. Створіть підкласи `PDFReport` і `ExcelReport`, які реалізують метод `generate()` для формування звітів у форматах PDF і Excel відповідно. Перевірте, як метод `generate()` працює для різних форматів звітів.

12. Створіть базовий клас `TaxCalculator` з методом `calculate_tax()`. Створіть підкласи `IncomeTaxCalculator` і `SalesTaxCalculator`, які реалізують метод `calculate_tax()` для розрахунку податку на доходи і податку з продажу відповідно. Перевірте, як метод `calculate_tax()` працює для різних типів податків.

Тема 24
Абстрактні класи

24.1. Поняття абстрактного класу.
24.2. Поняття про метаклас.
24.3. Інтерфейси.

24.1. Поняття абстрактного класу

Абстрактний клас – це клас, який має хоча б один абстрактний метод. Абстрактні методи – це методи, що оголошені в базовому класі, але не мають конкретної реалізації. Замість цього вони мають бути реалізовані в підкласах. Абстрактний клас не може бути інстанційований напряму, він використовується лише для визначення структури для інших класів, які успадковують його.

Для визначення абстрактного класу та методів у Python використовується модуль *abc* (*Abstract Base Classes*). Він надає декоратор *@abstractmethod*, який можна використовувати для позначення методів як абстрактних у базовому класі.

Основна мета абстрактних класів – це встановлення загального інтерфейсу для класів, що їх успадковують, і забезпечення, що ці класи реалізують певний набір методів. Це дає змогу створювати більш структуровану та прогнозовану архітектуру програм.

Отже, абстрактні класи є потужним інструментом для створення більш гнучких та розширюваних програм у Python.

1. *Shape* – базовий клас для геометричних фігур:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

rect = Rectangle(5, 4)
print(rect.area())  # Output: 20
```

У цьому прикладі *Shape* є базовим абстрактним класом, який визначає метод *area*, але не має конкретної реалізації. Клас *Rectangle* успадковує *Shape* та реалізовує метод *area*.

2. *Vehicle* – базовий клас для транспортних засобів:

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def move(self):
        pass

class Car(Vehicle):
    def move(self):
        print("Car is moving")

class Plane(Vehicle):
    def move(self):
        print("Plane is flying")

car = Car()
car.move()  # Output: Car is moving
```

У цьому прикладі *Vehicle* є базовим абстрактним класом з методом *move*, який не має реалізації. Класи *Car і Plane* успадковують *Vehicle* і реалізовують метод *move*.

3. *Animal* – базовий клас для тварин:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        print("Woof!")

class Cat(Animal):
    def sound(self):
        print("Meow!")

cat = Cat()
cat.sound()  # Output: Meow!
```

У цьому прикладі *Animal* є базовим абстрактним класом, який визначає метод *sound*, але не має конкретної реалізації. Класи *Dog* і *Cat* успадковують *Animal* і реалізовують метод *sound*.

4. DatabaseConnection – базовий клас для підключення до бази даних:

```
from abc import ABC, abstractmethod

class DatabaseConnection(ABC):
    @abstractmethod
    def connect(self):
        pass
```

```
class MySQLConnection(DatabaseConnection):
    def connect(self):
        print("Connecting to MySQL database")

class PostgreSQLConnection(DatabaseConnection):
    def connect(self):
        print("Connecting to PostgreSQL database")

connection = PostgreSQLConnection()
connection.connect() # Output: Connecting to PostgreSQL database
```

У цьому прикладі *DatabaseConnection* є базовим абстрактним класом з методом *connect*, який не має реалізації. Класи *MySQLConnection* і *PostgreSQLConnection* успадковують *DatabaseConnection* і реалізовують метод *connect*.

24.2. Поняття про метаклас

Метакласи в Python – це класи, які використовуються для створення інших класів. Вони надають можливість керувати створенням класів, їхнім поведінкою та властивостями. Метакласи є потужним інструментом для метапрограмування і використовуються рідко, але в деяких випадках можуть бути дуже корисними.

1. Створення метакласу для перевірки атрибутів класу:

```
class MyMeta(type):
    def __new__(cls, name, bases, attrs):
        if 'attr' not in attrs:
            raise AttributeError("Class must have attribute 'attr'")
        return super().__new__(cls, name, bases, attrs)

class MyClass(metaclass=MyMeta):
    attr = 42

# Приклад використання
# MyClass без атрибута attr викине AttributeError
# MyClass з атрибутом attr буде створено успішно
```

У цьому прикладі метаклас *MyMeta* перевіряє наявність атрибута *attr* у створюваному класі. Якщо атрибут не знайдено, метаклас викидає виняток *AttributeError*.

2. Створення метакласу для реєстрації класів:

```
class ClassRegistry(type):
    registry = {}

    def __new__(cls, name, bases, attrs):
        new_cls = super().__new__(cls, name, bases, attrs)
        cls.registry[name] = new_cls
        return new_cls

class MyClass1(metaclass=ClassRegistry):
    pass

class MyClass2(metaclass=ClassRegistry):
    pass

# Приклад використання
print(ClassRegistry.registry)
# Output: {'MyClass1': <class '__main__.MyClass1'>, 'MyClass2': <class '__main__.MyClass2'>}
```

У цьому прикладі метаклас *ClassRegistry* реєструє створені класи у словнику *registry*.

3. Створення метакласу для автоматичного надання атрибутів:

```
class AutoAttribute(type):
    def __new__(cls, name, bases, attrs):
        for attr_name, attr_value in attrs.items():
            if isinstance(attr_value, int):
                attrs[attr_name] = attr_value * 2
        return super().__new__(cls, name, bases, attrs)
```

```
class MyClass(metaclass=AutoAttribute):
    attr1 = 5
    attr2 = 10

# Приклад використання
print(MyClass.attr1)  # Output: 10
print(MyClass.attr2)  # Output: 20
```

У цьому прикладі метаклас *AutoAttribute* автоматично подвоює значення атрибутів класу, якщо вони є цілими числами.

4. Створення метакласу для додавання методів:

```
class MethodInjector(type):
    def __new__(cls, name, bases, attrs):
        def new_method(self):
            return "Injected method"

        attrs['new_method'] = new_method
        return super().__new__(cls, name, bases, attrs)

class MyClass(metaclass=MethodInjector):
    pass

# Приклад використання
obj = MyClass()
print(obj.new_method())  # Output: Injected method
```

У цьому прикладі метаклас *MethodInjector* додає новий метод *new_method* до класу *MyClass*.

5. Створення метакласу для створення *Singleton*:

```
class Singleton(type):
    _instances = {}

    def __call__(cls, *args, kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, kwargs)
        return cls._instances[cls]

class MyClass(metaclass=Singleton):
    pass

# Приклад використання
obj1 = MyClass()
obj2 = MyClass()
print(obj1 is obj2)  # Output: True
```

У цьому прикладі метаклас *Singleton* забезпечує, що тільки один екземпляр класу *MyClass* буде створено, і всі подальші виклики будуть повертати цей один екземпляр.

24.3. Інтерфейси

У Python немає вбудованої підтримки інтерфейсів, як у багатьох інших мов програмування, таких як Java або C#. Однак можна створювати імітацію інтерфейсів за допомогою абстрактних класів або абстрактних методів.

1. Інтерфейс *"Форма"* з методом *"вивести"* та *"очистити"*:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def draw(self):
        pass
```

```
    @abstractmethod
    def clear(self):
        pass

class Circle(Shape):
    def draw(self):
        print("Drawing Circle")

    def clear(self):
        print("Clearing Circle")

# Приклад використання
circle = Circle()
circle.draw()  # Output: Drawing Circle
circle.clear()  # Output: Clearing Circle
```

2. Інтерфейс *"Звір"* з методами *"їсти"* та *"спати":*

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def eat(self):
        pass

    @abstractmethod
    def sleep(self):
        pass

class Dog(Animal):
    def eat(self):
        print("Dog is eating")

    def sleep(self):
        print("Dog is sleeping")
```

```
# Приклад використання
dog = Dog()
dog.eat()  # Output: Dog is eating
dog.sleep()  # Output: Dog is sleeping
```

Ці приклади показують, як можна використовувати абстрактні класи та методи для створення імітації інтерфейсів у Python. Інтерфейси допомагають забезпечити спільний інтерфейс для класів, які реалізовують певний набір функціональностей, що полегшує роботу з кодом та його розширення.

Перелік питань для самоконтролю

1. Абстрактний клас.
2. Абстрактні методи.
3. Поняття про метакласи.
4. Інтерфейси.

Тестові питання

1. Що таке абстрактний клас у Python:

а) клас, який можна інстанціювати безпосередньо;

б) клас, який має хоча б один абстрактний метод;

в) клас, що не має жодних методів;

г) клас, що визначає структуру для підкласів, але не може бути інстанційований напряму.

2. Яка бібліотека використовується для роботи з абстрактними класами в Python:

а) os;

б) math;

в) abc;

г) numpy.

3. Який декоратор використовується для визначення абстрактного методу в класі:

а) @abstractmethod;

б) @abstract;

в) @abstractmethod;

г) @method.

4. Що станеться, якщо спробувати створити екземпляр абстрактного класу без реалізації всіх абстрактних методів:

а) помилка виконання (RuntimeError);

б) помилка компіляції (SyntaxError);

в) клас буде створений, але буде неможливо викликати його методи;

г) нічого, клас буде створений без будь-яких проблем.

5. Які з наведених нижче не є прикладом використання абстрактних класів у Python:

а) визначення загального інтерфейсу для класів;

б) використання абстрактних методів для задання обов'язкових операцій;

в) визначення статичних методів;

г) використання абстрактних класів для спадкування.

6. Що таке спадкування абстрактного класу в Python:

а) можливість класу мати більше одного батьківського класу;

б) передача атрибутів та методів з батьківського класу до дочірнього;

в) визначення абстрактних методів у батьківському класі, які дочірні класи повинні реалізувати;

г) використання класів-домішок.

7. Що таке абстрактний метод у Python:

а) метод, який завжди повертає None;

б) метод, який може бути перевизначений у дочірньому класі;

в) метод, який визначає інтерфейс класу, але не має реалізації;

г) метод, який завжди викликає помилку.

8. Що буде виведено в результаті виконання коду:

```python
from abc import ABC, abstractmethod

class Parent(ABC):
    @abstractmethod
    def method(self):
        pass

class Child(Parent):
    pass

obj = Child()
print(isinstance(obj, Parent))
```

а) True;
б) False;
в) TypeError;
г) AttributeError.

9. Які з наведених нижче не є перевагою використання абстрактних класів у Python:

а) забезпечення статичної типізації;
б) гарантує виконання обов'язкових операцій у підкласах;
в) визначення загального інтерфейсу для класів;
г) спрощення управління кодом та зниження кількості помилок.

10. Що таке модуль abc у Python:

а) модуль, який надає базові класи для створення абстрактних класів та методів;
б) модуль для арифметичних операцій;
в) модуль для роботи зі строками;
г) модуль для роботи з файлами.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Створіть абстрактний клас `Shape` з абстрактним методом `area()`. Реалізуйте підкласи `Circle` і `Rectangle`, які реалізують метод `area()` для обчислення площі відповідно.

2. Створіть абстрактний клас `Message` з абстрактним методом `send()`. Реалізуйте підкласи `EmailMessage` і `SMSMessage`, які реалізують метод `send()` для надсилання електронних листів і SMS повідомлень.

3. Створіть абстрактний клас `Vehicle` з абстрактним методом `start_engine()`. Реалізуйте підкласи `Car` і `Motorcycle`, які реалізують метод `start_engine()` для запуску двигуна автомобіля і мотоцикла.

4. Створіть абстрактний клас `Storage` з абстрактним методом `store()`. Реалізуйте підкласи `FileStorage` і `DatabaseStorage`, які реалізують метод `store()` для зберігання даних у файлі й базі даних.

5. Створіть абстрактний клас `EventHandler` з абстрактним методом `handle()`. Реалізуйте підкласи `ClickEventHandler` і `KeyPressEventHandler`, які реалізують метод `handle()` для обробки натискання кнопок миші й клавіатури.

6. Створіть абстрактний клас `NumberProcessor` з абстрактним методом `process()`. Реалізуйте підкласи `IntegerProcessor` і `FloatProcessor`, які реалізують метод `process()` для обробки цілих чисел і чисел з плаваючою комою відповідно.

7. Створіть абстрактний клас `Report` з абстрактним методом `generate()`. Реалізуйте підкласи `PDFReport` і `ExcelReport`, які реалізують метод `generate()` для формування звітів у форматах PDF і Excel.

8. Створіть абстрактний клас `GraphicalElement` з абстрактним методом `draw()`. Реалізуйте підкласи `Circle` і `Rectangle`, які реалізують метод `draw()` для малювання кола й прямокутника.

9. Створіть абстрактний клас `Operation` з абстрактним методом `compute()`. Реалізуйте підкласи `Addition` і `Subtraction`, які реалізують метод `compute()` для виконання відповідних арифметичних операцій.

10. Створіть абстрактний клас `Formatter` з абстрактним методом `format()`. Реалізуйте підкласи `UpperCaseFormatter` і `LowerCaseFormatter`, які реалізують метод `format()` для перетворення рядка в верхній і нижній регістр відповідно.

11. Створіть абстрактний клас `TaxCalculator` з абстрактним методом `calculate_tax()`. Реалізуйте підкласи `IncomeTaxCalculator` і `SalesTaxCalculator`, які реалізують метод `calculate_tax()` для розрахунку податку на доходи й податку з продажу.

12. Створіть абстрактний клас `Statistics` з абстрактним методом `calculate()`. Реалізуйте підкласи `MeanCalculator` і `MedianCalculator`, які реалізують метод `calculate()` для обчислення середнього значення і медіани відповідно.

13. Створіть абстрактний клас `FileAnalyzer` з абстрактним методом `analyze()`. Реалізуйте підкласи `TextFileAnalyzer` і `BinaryFileAnalyzer`, які реалізують метод `analyze()` для аналізу текстових і бінарних файлів відповідно.

14. Створіть абстрактний клас `RequestHandler` з абстрактним методом `handle_request()`. Реалізуйте підкласи `GETRequestHandler` і `POSTRequestHandler`, які реалізують метод `handle_request()` для обробки GET і POST запитів відповідно.

15. Створіть інтерфейс `Shape` з методами `area()` і `perimeter()`. Реалізуйте класи `Circle` та `Rectangle`, які імплементують цей інтерфейс для обчислення площі й периметра відповідно.

16. Створіть інтерфейс `DataProcessor` з методом `process(data)`. Реалізуйте класи `CSVProcessor` і `JSONProcessor`, які імплементують цей інтерфейс для обробки даних у форматах CSV і JSON відповідно.

17. Створіть інтерфейс `OrderManager` з методами `create_order(order_details)` і `cancel_order(order_id)`. Реалізуйте класи `OnlineOrderManager` і `InStoreOrderManager`, які імплементують цей інтерфейс для управління онлайн і в магазині замовленнями відповідно.

18. Створіть інтерфейс `MessageSender` з методом `send_message(message)`. Реалізуйте класи `EmailSender` і `SMSSender`, які імплементують цей інтерфейс для надсилання повідомлень електронної пошти і SMS відповідно.

19. Створіть інтерфейс `Storage` з методами `save(data)` і `load()`. Реалізуйте класи `FileStorage` і `DatabaseStorage`, які імплементують цей інтерфейс для зберігання даних у файлі й базі даних відповідно.

20. Створіть інтерфейс `Vehicle` з методами `start_engine()` і `stop_engine()`. Реалізуйте класи `Car` і `ElectricScooter`, які імплементують цей інтерфейс для запуску та зупинки двигуна автомобіля й електросамоката відповідно.

21. Створіть абстрактний клас `Form` з абстрактними методами `area()` і `perimeter()`. Реалізуйте підкласи `Circle` і `Rectangle`, які реалізують ці методи для обчислення площі та периметра відповідно.

22. Створіть абстрактний клас `User` з абстрактними методами `login()` і `logout()`. Реалізуйте підкласи `AdminUser` і `RegularUser`, які реалізують ці методи для входу й виходу в систему.

23. Створіть абстрактний клас `DataHandler` з абстрактними методами `read()` і `write(data)`. Реалізуйте підкласи `CSVHandler` і `JSONHandler`, які реалізують ці методи для читання й запису даних у відповідні формати.

24. Створіть абстрактний клас `Vehicle` з абстрактними методами `start_engine()` і `stop_engine()`. Реалізуйте підкласи `Car` і `Bike`, які реалізують ці методи для запуску та зупинки двигуна відповідно.

25. Створіть абстрактний клас `Report` з абстрактним методом `generate()`. Реалізуйте підкласи `PDFReport` і `ExcelReport`, які реалізують цей метод для формування звітів у форматах PDF та Excel відповідно.

26. Створіть абстрактний клас `RequestHandler` з абстрактним методом `handle_request(request)`. Реалізуйте підкласи `GETRequestHandler` і `POSTRequestHandler`, які реалізують цей метод для обробки GET та POST запитів відповідно.

Тема 25
Робота з файлами в Python

25.1. Основні поняття про файл і типи файлів.

25.2. Відкриття файлу та формат функції open( ).

25.3. Абсолютний та відносний шлях до файлу.

25.4. Модифікатори відкриття файлу.

25.5. Особливості роботи з механізмом буферизації файлів.

25.6. Поняття про кодування текстових файлів та особливості роботи з різними кодуваннями.

25.7. Методи для роботи з файлами.

25.8. Функції для маніпулювання файлами.

25.9. Перевірка наявності файлу, розміру файлу, часу останнього доступу, часу створення, часу останньої зміни.

25.10. Збереження об’єктів у файлі.

25.11. Функції модуля pickle для роботи з файлами.

25.12. Використання модуля shelve для зберігання даних у файлі за ключем.

25.13. Приклади роботи з файлами.

25.1. Основні поняття про файл і типи файлів

Файл – це носій інформації, який зберігається на зовнішньому пристрої (наприклад, на жорсткому диску, флеш-накопичувачі, *CD-ROM* тощо) і містить дані, такі як текст, зображення, аудіо, відео тощо. У Python робота з файлами відбувається за допомогою вбудованих функцій та об’єктів модуля *io*.

Основні поняття про файли:

1. Ім’я файлу. Це унікальна назва, яка вказує на місце зберігання файлу у файловій системі.

2. Розширення файлу. Це частина імені файлу, яка зазвичай вказує на тип даних, що зберігаються у файлі. Наприклад, у файлі з розширенням *.txt* зазвичай зберігається текстова інформація.

3. Шлях до файлу. Це повний або відносний шлях до файлу у файловій системі, охоплюючи ім’я файлу та всі необхідні папки.

4. Режим відкриття файлу. Це параметр, який вказує на те, як саме буде відкритий файл – для читання, запису, додавання тощо. Наприклад, `*r*` для читання, `*w*` для запису, `*a*` для додавання.

Типи файлів:

1. Текстові файли. Це файли, що містять текстову інформацію. Їх можна відкривати та редагувати за допомогою текстових редакторів. Наприклад, файли з розширенням *.txt*.

2. Бінарні файли. Це файли, що містять нетекстову інформацію, таку як зображення, аудіо, відео тощо. Їх неможливо прочитати або редагувати за допомогою текстових редакторів. Наприклад, файли з розширеннями *.jpg, .mp3, .mp4*.

3. Текстово-бінарні файли. Це файли, які поєднують у собі текстові та бінарні дані. Наприклад, *HTML*-файли, які містять текстову інформацію та посилання на зображення або інші бінарні дані.

4. Текстові файли з розділенням. Це текстові файли, в яких дані розділені за допомогою спеціального символу або символів, таких як кома чи табуляція. Наприклад, *CSV*-файли, *JSON*-файли.

5. Архіви. Це спеціальні типи файлів, які містять у собі інші файли та теки, зазвичай стиснуті для зменшення розміру. Наприклад, файли з розширеннями *.zip*, *.tar.gz*.

25.2. Відкриття файлу та формат функції open( )

Функція *open()* у Python використовується для відкриття файлів у різних режимах. Вона приймає два основних аргументи: шлях до файлу та режим відкриття. Загальний формат використання функції *open()*:

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)

Параметри:

- *file*: Шлях до файлу або об’єкт файлового дескриптора.
- *mode*: Режим відкриття файлу. Може бути одним з таких:
- '*r*': Читання (за замовчуванням). Файл повинен існувати.

- '*w*': Запис. Якщо файл не існує, він буде створений. Якщо файл існує, вміст буде замінений.

- '*x*': Ексклюзивне створення. Ті самі правила, що й `'w'`, але якщо файл уже існує, операція завершиться помилкою.

- '*a*': Додавання (дописування) в кінець файлу. Новий вміст буде додано до кінця файлу.

- '*b*': Бінарний режим.

- '*t*': Текстовий режим (за замовчуванням).

- '+': Оновлення (читання і запис).

- Інші параметри, такі як *buffering, encoding, errors, newline, closefd, opener*, є необов'язковими і використовуються для зміни стандартної поведінки функції *open()*.

```
# Відкриття текстового файлу для читання
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)

# Відкриття бінарного файлу для запису
with open('example.bin', 'wb') as file:
    file.write(b'Hello, binary world!')

# Відкриття файлу для дописування
with open('example.txt', 'a') as file:
    file.write('\nAdditional line')

# Відкриття файлу для читання та запису
with open('example.txt', 'r+') as file:
    content = file.read()
    print(content)
    file.write('\nUpdating the file')
```

Функція *open()* дає змогу зчитувати, записувати, додавати до файлу або працювати з ним у бінарному режимі залежно від обраного режиму відкриття. За допомогою контекстного менеджера *with* файл автоматично закриється після виходу з блоку *with*, що дає змогу уникнути проблем з витіканням ресурсів.

25.3. Абсолютний та відносний шлях до файлу

У Python існують два типи шляхів до файлів: абсолютний і відносний.

1. Абсолютний шлях до файлу – це повний шлях до файлу від кореневої директорії файлової системи. Наприклад, в *UNIX*-подібних операційних системах (таких як *LINUX, MACOS*) абсолютний шлях може виглядати як */home/user/example.txt*, де */home/user/* – це коренева директорія. У *WINDOWS* абсолютний шлях може бути, наприклад, *c:\users\user\example.txt*, де *c:* – це кореневий диск.

2. Відносний шлях до файлу – це шлях до файлу відносно поточної робочої директорії, в якій ви виконуєте Python-скрипт або команда. Наприклад, якщо поточна робоча директорія має шлях */home/user/*, то відносний шлях *example.txt* буде вказувати на файл */home/user/example.tx*`. Відносний шлях може також використовувати `..` для вказівки на батьківську директорію. Наприклад, якщо ви перебуваєте в */home/user/subdir/*, то *../example.txt* буде вказувати на файл */home/user/example.txt*.

```
# Абсолютний шлях
absolute_path = '/home/user/example.txt'

# Відносний шлях
relative_path = 'example.txt'
```

Зазвичай абсолютні шляхи використовуються там, де потрібна точність і незалежність від поточної робочої директорії, наприклад, під час роботи з файлами за межами поточної програми. Відносні шляхи зручніші для використання у випадках, коли ви працюєте в межах однієї структури каталогів і не хочете змінювати власність файлів під час переміщення вашої програми на інший комп’ютер.

25.4. Модифікатори відкриття файлу

У Python модифікатори відкриття файлу вказують на режими, в яких файл може бути відкритий. Вони визначають, чи можна читати, записувати або дописувати в файл, а також чи може файл бути відкритий у текстовому або бінарному режимі.

1. *`r` (read)* – відкриває файл для читання. Файл повинен існувати, інакше виникне помилка. Це режим за замовчуванням, якщо ви не вказуєте інший. Використовується для читання текстових або бінарних файлів.

2. *`w` (write)* – відкриває файл для запису. Якщо файл не існує, він буде створений. Якщо файл уже існує, його вміст буде замінений. Використовується для запису в текстові або бінарні файли.

3. *`a` (append)* – відкриває файл для дописування. Новий вміст буде доданий до кінця файлу без заміни існуючого вмісту. Якщо файл не існує, він буде створений. Використовується для дописування в текстові або бінарні файли.

4. *`x` (exclusive creation)* – відкриває файл для ексклюзивного створення. Якщо файл уже існує, виникає помилка. Використовується для створення нового файлу, якщо файл з указаним ім’ям ще не існує.

5. *`r+` (read/write)* – відкриває файл для читання та запису. Файл повинен існувати. Використовується для зчитування та записування в текстові або бінарні файли.

6. *`w+` (read/write)* – відкриває файл для читання та запису. Якщо файл не існує, він буде створений. Якщо файл уже існує, його вміст буде замінений.

7. *`a+` (read/append)* – відкриває файл для читання та дописування. Новий вміст буде доданий до кінця файлу без заміни існуючого вмісту. Якщо файл не існує, він буде створений.

8. *`b` (binary mode)* – відкриває файл у бінарному режимі. Використовується, коли потрібно працювати з бінарними даними, такими як зображення або аудіо-файли.

25.5. Особливості роботи з механізмом буферизації файлів

Механізм буферизації файлів у Python використовується для збереження даних у внутрішньому буфері перед їх записом у файл або після їх зчитування з файлу. Це дає змогу зменшити кількість взаємодій з файловою системою, що може покращити продуктивність програми. Особливості роботи з механізмом буферизації файлів охоплюють таке:

1. Автоматична буферизація. Багато операцій вводу/виводу, такі як *read()*, *write(), readline()* автоматично використовують буферизацію для оптимізації роботи з файлами. Це означає, що дані спочатку зберігаються у внутрішньому буфері, а потім записуються у файл.

2. Мануальна буферизація. Ви можете вказати розмір буфера, використовуючи аргумент *buffering* під час відкриття файлу за допомогою функції *open()*. Значення `-1` вказує на автоматичну буферизацію, `0` вказує на відсутність буферизації, а позитивне ціле число вказує на розмір буфера в байтах.

3. Методи *flush()* та *close()*. Метод *flush()* використовується для виведення всіх даних з буфера до файлу, зберігаючи файл в актуальному стані. Метод *close()* автоматично викликає метод *flush()* перед закриттям файлу, що гарантує, що всі дані з буфера будуть записані до файлу перед його закриттям.

4. Вплив на продуктивність. Використання буферизації може покращити продуктивність програми, оскільки зменшується кількість взаємодій з файловою системою, що може бути відносно повільним процесом.

5. Буферизація вводу та виводу. Буферизація може використовуватися як для оптимізації читання з файлу, так і для оптимізації запису в файл.

```
with open('example.txt', 'w', buffering=1024) as file:
    file.write('Hello, buffer!')
```

У цьому прикладі буфер розміром 1024 байти буде використовуватися для збереження даних перед записом їх до файлу.

25.6. Поняття про кодування текстових файлів та особливості роботи з різними кодуваннями

Кодування текстових файлів визначає спосіб представлення символів у файлі за допомогою числових значень (байтів). Оскільки комп'ютери працюють з числами, а не зі звичайними символами, кодування використовується для перетворення символів у послідовності байтів, які можна зберігати й обробляти. Основні поняття та особливості роботи з кодуваннями включають таке:

1. *Unicode* – це стандарт кодування, що містить у собі символи практично всіх письмових мов світу, включаючи латиницю, кирилицю, китайську та її ієрогліфи, японську і багато інших. *Unicode* визначає унікальний номер для кожного символу та використовує різні кодування, такі як *UTF-8, UTF-16, UTF-32*, для представлення цих символів у вигляді байтів.

2. *UTF-8* – це найпопулярніше та універсальне кодування, яке використовується для представлення *Unicode* символів у вигляді байтів. У *UTF-8* кожен символ представлений від 1 до 4 байтів, залежно від його кодування.

3. *UTF-16* – це кодування *Unicode*, в якому кожен символ представлений 2 або 4 байтами. *UTF-16* частіше використовується в системах *Windows*.

4. *UTF-32* – це кодування *Unicode*, в якому кожен символ представлений точно 4 байтами. *UTF-32* забезпечує простий зв'язок між кодуванням та номером символу, але займає більше місця порівняно з іншими кодуваннями.

5. *ANSI (Windows-1252)* – це кодування, яке використовується в системах *Windows* для представлення тексту в старіших версіях операційної системи. Він підтримує лише обмежену множину символів та має обмежену міжнародну підтримку.

6. *ASCII* – це стандарт кодування, який використовується для представлення англійського алфавіту та основних спеціальних символів у вигляді числових значень (байтів). *ASCII* кодування не підтримує міжнародні символи.

В процесі роботи з текстовими файлами важливо вказувати правильне кодування під час їх відкриття та збереження. Наприклад, у Python ви можете відкрити файл у певному кодуванні так:

```
with open('example.txt', 'r', encoding='utf-8') as file:
    content = file.read()
    print(content)
```

У цьому прикладі файл `example.txt` відкривається для читання в кодуванні *UTF-8*. Такий підхід дає змогу коректно зчитувати та записувати текстові дані у файл у будь-якій мові з урахуванням їхніх специфічних символів.

25.7. Методи для роботи з файлами

У Python існує кілька вбудованих методів для роботи з файлами.

1. *open()* – використовується для відкриття файлу. Він приймає ім’я файлу та режим відкриття.

```
file = open("example.txt", "r")
```

2. *read(size)* – використовується для зчитування вмісту файлу. Він приймає необов’язковий параметр *size*, який указує кількість байтів, які потрібно зчитати. Якщо параметр *size* не вказано, зчитується весь вміст файлу.

```
content = file.read()
```

3. *readline(size)* – метод зчитує рядок з файлу. Він також приймає необов’язковий параметр *size*, щоб вказати кількість байтів для зчитування.

```
line = file.readline()
```

4. *readlines()* – метод зчитує всі рядки з файлу й повертає їх у вигляді списку рядків.

```
lines = file.readlines()
```

5. *write(string)* – використовується для запису рядка у файл. Приклад:

```
file.write("Hello, world!")
```

6. *writelines(lines)* – використовується для запису списку рядків у файл.

```
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
file.writelines(lines)
```

7. *close()* – використовується для закриття файлу.

```
file.close()
```

8. *flush()* – цей метод використовується для виведення буферизованих даних на диск. Приклад:

```
file.flush()
```

Це лише декілька основних методів для роботи з файлами в Python. Існує багато інших методів, які можуть бути корисними залежно від вашого конкретного використання, таких як переміщення вказівника у файлі, перевірка чи файл закритий тощо.

25.8. Функції для маніпулювання файлами

У Python існують різні вбудовані функції, які дають змогу маніпулювати файлами.

1. *os.path.exists(path)* – ця функція перевіряє, чи існує файл або директорія за вказаним шляхом. Повертає *True*, якщо файл або директорія існує, і *False* в іншому випадку.

```
import os

if os.path.exists("example.txt"):
    print("Файл існує")
else:
    print("Файл не існує")
```

2. *os.path.isfile(path)* – ця функція перевіряє, чи є вказаний шлях файлом. Повертає *True*, якщо файл існує, і він є файлом, і *False* в іншому випадку.

```
import os

if os.path.isfile("example.txt"):
    print("Це файл")
else:
    print("Це не файл")
```

3. *os.path.isdir(path)* – ця функція перевіряє, чи є вказаний шлях директорією. Повертає *True*, якщо директорія існує, і він є директорією, і *False* в іншому випадку.

```
import os

if os.path.isdir("my_directory"):
    print("Це директорія")
else:
    print("Це не директорія")
```

4. *os.rename(old, new)* – ця функція використовується для перейменування файлу або директорії. Вона приймає старий шлях та новий шлях.

```
import os

os.rename("old_name.txt", "new_name.txt")
```

5. *os.remove(path)* – ця функція використовується для видалення файлу.

```
import os

os.remove("example.txt")
```

6. *os.mkdir(path)* – створює нову директорію за вказаним шляхом.

```
import os

os.mkdir("new_directory")
```

7. *os.rmdir(path)* – видаляє порожню директорію.

```
import os

os.rmdir("empty_directory")
```

8. *shutil.move(src, dst)* – ця функція використовується для переміщення файлів або директорій.

```
import shutil

shutil.move("source.txt", "destination_folder")
```

Це лише декілька прикладів функцій для маніпулювання файлами в Python. Існує ще багато інших функцій, які можна використовувати для роботи з файлами, залежно від вашого конкретного використання.

25.9. Перевірка наявності файлу, розміру файлу, часу останнього доступу, часу створення, часу останньої зміни

У Python для отримання різноманітної інформації про файл, такої як перевірка наявності файлу, його розміру та різні часові показники, можна використовувати модуль *os* або *os.path*.

1. Перевірка наявності файлу:

```
import os

if os.path.exists("example.txt"):
   print("Файл існує")
else:
   print("Файл не існує")
```

2. Отримання розміру файлу (у байтах):

```
import os

size = os.path.getsize("example.txt")
print("Розмір файлу:", size, "байт")
```

3. Отримання часу останнього доступу до файлу:

```
import os
import datetime

timestamp = os.path.getatime("example.txt")
last_access_time = datetime.datetime.fromtimestamp(timestamp)
print("Час останнього доступу до файлу:", last_access_time)
```

4. Отримання часу створення файлу:

```
import os
import datetime

timestamp = os.path.getctime("example.txt")
creation_time = datetime.datetime.fromtimestamp(timestamp)
print("Час створення файлу:", creation_time)
```

5. Отримання часу останньої зміни файлу:

```
import os
import datetime

timestamp = os.path.getmtime("example.txt")
last_modify_time = datetime.datetime.fromtimestamp(timestamp)
print("Час останньої зміни файлу:", last_modify_time)
```

Ці функції дають змогу отримати різноманітну інформацію про файл, що може бути корисною для різних операцій обробки файлів у вашій програмі.

25.10. Збереження об’єктів у файлі

У Python для збереження об’єктів у файлі можна використовувати модуль *pickle*. *pickle* дає змогу серіалізувати об’єкти Python у байтовий потік, який потім можна зберегти у файлі.

```
import pickle

# Припустимо, ми маємо словник, який ми хочемо зберегти у файлі
data = {'key': 'value', 'name': 'John', 'age': 30}

# Відкриваємо файл у режимі запису байтів
with open('data.pkl', 'wb') as f:
    # Використовуємо функцію pickle.dump() для збереження об'єкта у файлі
    pickle.dump(data, f)
```

Цей код створить файл з назвою *data.pkl*, в який буде збережений словник *data*.

Щоб завантажити об’єкт з файлу, ви можете використати функцію *pickle.load()*.

```
import pickle

# Відкриваємо файл у режимі читання байтів
with open('data.pkl', 'rb') as f:
    # Використовуємо функцію pickle.load() для завантаження об'єкта з файлу
    loaded_data = pickle.load(f)

print(loaded_data)
```

Цей код завантажить об’єкт з файлу *data.pkl* і виведе його на екран. Ви отримаєте той самий словник, який ви зберегли раніше.

Використання *pickle* може бути дуже корисним для збереження складних об’єктів Python, таких як списки, словники, класи тощо, у файлі. Однак будьте обережні, коли використовуєте *pickle*, особливо коли ви працюєте з файлами, які можуть бути використані іншими користувачами або програмами, оскільки він може виконувати відновлення об’єктів з потенційно небезпечних джерел.

25.11. Функції модуля pickle для роботи з файлами

Модуль *pickle* у Python надає декілька функцій для роботи з файлами для серіалізації та десеріалізації об’єктів.

1. *pickle.dump(obj, file, protocol=None, *, fix_imports=True)* – ця функція серіалізує об'єкт *obj* у файл, вказаний об’єктом файлу *file*. Параметр *protocol* вказує версію протоколу для використання (за замовчуванням використовується найновіший доступний). Якщо параметр *fix_imports* встановлено в *True*, то старі структури модулів будуть коректно відображені в назви модулів; якщо встановлено в *False*, то старі структури модулів будуть представлені як _ _*main*_ _.

```
import pickle

data = {'key': 'value', 'name': 'John', 'age': 30}
with open('data.pkl', 'wb') as f:
    pickle.dump(data, f)
```

2. *pickle.load(file, *, fix_imports=True, encoding="ASCII", errors="strict")* – ця функція десеріалізує об'єкт з файлу, вказаного об'єктом файлу *file*, та повертає його. Параметри *fix_imports, encoding і errors* аналогічні до функції *pickle.dump()*.

```
import pickle

with open('data.pkl', 'rb') as f:
    loaded_data = pickle.load(f)
```

3. *pickle.dumps(obj, protocol=None, *, fix_imports=True)* – серіалізує об'єкт *`obj`* у рядок байтів. Параметри *`protocol` і `fix_imports`* аналогічні до функції *`pickle.dump()`*, але результатом є об'єкт *bytes*. Приклад:

```
import pickle

data = {'key': 'value', 'name': 'John', 'age': 30}
bytes_data = pickle.dumps(data)
```

4. *pickle.loads(bytes_object, *, fix_imports=True, encoding="ASCII", errors="strict")*. Ця функція десеріалізує об'єкт з рядка байтів *bytes_object* і повертає його. Параметри *fix_imports, encoding і errors* аналогічні до функції *pickle.load()*.

```
import pickle

loaded_data = pickle.loads(bytes_data)
```

Ці функції дають змогу легко зберігати та відновлювати об'єкти Python у файлах або в рядках байтів за допомогою модуля *pickle*.

25.12. Використання модуля shelve для зберігання даних у файлі за ключем

Модуль *shelve* в Python дає змогу зберігати об’єкти Python у файлі, використовуючи ключі для доступу до них. Це подібно до словника, але дані автоматично зберігаються в файлі.

```
import shelve

# Відкриваємо файл shelve в режимі запису
with shelve.open('mydata') as shelf:
    # Зберігаємо дані за ключем
    shelf['key1'] = {'name': 'John', 'age': 30}
    shelf['key2'] = [1, 2, 3, 4, 5]

# Відкриваємо файл shelve у режимі читання
with shelve.open('mydata') as shelf:
    # Отримуємо дані за ключем
    data1 = shelf['key1']
    data2 = shelf['key2']

print(data1)
print(data2)
```

У цьому прикладі ми спочатку відкриваємо файл *mydata* за допомогою *shelve.open()* у режимі запису та зберігаємо дані за ключами *key1* та *key2*. Потім ми закриваємо файл. Потім ми знову відкриваємо той самий файл у режимі читання та отримуємо дані за допомогою ключів.

Модуль *shelve* автоматично зберігає дані у файлі, використовуючи формат, який він сам визначає. Можна зберігати будь-які об’єкти Python, які можна серіалізувати (наприклад, словники, списки, кортежі, класи тощо). Цей модуль є зручним для зберігання невеликих об’єктів даних у файлах та забезпечує простий доступ до них за допомогою ключів.

25.13. Приклади роботи з файлами

1. Відкриття текстового файла. У Python відкриття текстового файлу зазвичай виконується за допомогою вбудованої функції *open()*.

```
file_path = "example.txt"

# Відкриття текстового файлу для читання
with open(file_path, "r") as file:
    # Читаємо вміст файлу
    content = file.read()

print(content)
```

У цьому прикладі:

- *file_path* – це шлях до файлу, який ви хочете відкрити;
- *"r"* – це режим відкриття файлу, вказує на режим читання. Файл буде відкрито для читання, і якщо файл не існує, виникне помилка.

Використання конструкції *with open() as file* гарантує автоматичне закриття файлу після виконання блоку коду всередині *with*.

2. Запис даних у текстові файли. Запис даних у текстовий файл у Python виконується за допомогою вбудованої функції *open()* у режимі запису.

```
file_path = "example.txt"

# Відкриваємо текстовий файл для запису
with open(file_path, "w") as file:
    # Записуємо дані у файл
    file.write("Hello, world!\n")
    file.write("This is a new line.")
```

У цьому прикладі:

- *file_path* – це шлях до файлу, в який ви хочете записати дані;

- *"w"* – це режим відкриття файлу, вказує на режим запису. Якщо файл не існує, він буде створений. Якщо файл існує, його попередній вміст буде втрачено.

Важливо зауважити, що кожен виклик методу *write()* додасть дані до кінця файлу. Щоб додати новий рядок, ми використовуємо символ переносу рядка *\n*.

Якщо потрібно дописати дані у файл, не втрачаючи попередній вміст, можна відкрити файл у режимі допису, використовуючи *"a":*

```
file_path = "example.txt"

# Відкриваємо текстовий файл для допису
with open(file_path, "a") as file:
    # Записуємо нові дані у файл
    file.write("\nThis is appended data.")
```

3. Зчитування даних з текстових файлів. У Python зчитування даних з текстового файлу можна здійснити за допомогою кількох методів, таких як *read()*, *readline()* або ітерація через файловий об’єкт.

```
# Зчитування всього вмісту файлу за один раз за допомогою методу `read()`:
file_path = "example.txt"

# Відкриваємо текстовий файл для читання
with open(file_path, "r") as file:
    # Зчитуємо весь вміст файлу
    content = file.read()

print(content)
```

```
# Зчитування по одному рядку за раз за допомогою методу `readline()`:
file_path = "example.txt"

# Відкриваємо текстовий файл для читання
with open(file_path, "r") as file:
    # Зчитуємо перший рядок
    line1 = file.readline()
    # Зчитуємо другий рядок
    line2 = file.readline()

print("Перший рядок:", line1)
print("Другий рядок:", line2)

# Ітерація через файловий об'єкт для зчитування кожного рядка:
file_path = "example.txt"

# Відкриваємо текстовий файл для читання
with open(file_path, "r") as file:
    # Ітеруємося через файл і друкуємо кожен рядок
    for line in file:
        print(line.strip())
# Метод strip() видаляє пробіли та символи нового рядка з кінця рядка
```

4. Структуровані текстові файли: *CSV, XML, JSON*. Структуровані тексові файли, такі як *CSV, XML та JSON*, є популярними форматами для зберігання даних, які можуть бути легко прочитані іншими програмами або людьми.

CSV (Comma-Separated Values) – це формат збереження даних, де кожен рядок являє собою запис, а значення розділені комами. *CSV* файли можна легко створювати та читати за допомогою модуля *csv* у Python.

Приклад структури *CSV* файлу:

```
Name, Age, City
John, 30, New York
Alice, 25, Los Angeles
```

XML (eXtensible Markup Language) – це мова розмітки, яка використовує теги для представлення даних у структурованій формі. *XML* файли зазвичай використовуються для обміну даними між програмами.

Приклад структури XML файлу:

```
```xml
<people>
  <person>
    <name>John</name>
    <age>30</age>
    <city>New York</city>
  </person>
  <person>
    <name>Alice</name>
    <age>25</age>
    <city>Los Angeles</city>
  </person>
</people>
```

JSON (JavaScript Object Notation) – це легкий формат обміну даними, який використовується для представлення структурованих даних. Використовується для зберігання конфігурацій, обміну даними між сервером та клієнтом тощо.

Приклад структури JSON файлу:

```
[
  {"name": "John", "age": 30, "city": "New York"},
  {"name": "Alice", "age": 25, "city": "Los Angeles"}
]
```
```

У Python для роботи з цими форматами є вбудовані модулі:

- *csv* для роботи з *CSV* файлами;
- *xml.etree.ElementTree* або *lxml* для роботи з *XML* файлами;
- *json* для роботи з *JSON* файлами.

5. Перетворення відносного шляху в абсолютний.

У Python модуль *os.path* містить функцію *abspath()*, яка дає змогу перетворити відносний шлях у абсолютний.

```
import os

# Відносний шлях до файлу
relative_path = "folder/subfolder/file.txt"

# Перетворення відносного шляху в абсолютний
absolute_path = os.path.abspath(relative_path)

print("Відносний шлях:", relative_path)
print("Абсолютний шлях:", absolute_path)
```

Цей код перетворить відносний шлях *"folder/subfolder/file.txt"* у відповідний абсолютний шлях на основі поточної робочої директорії і виведе обидва шляхи на екран.

Функція *abspath()* розрізняє між абсолютними та відносними шляхами, тому вона може перетворити будь-який відносний шлях у відповідний абсолютний.

6. Можливі задавання шляхів до файлу та їх модифікація.

У Python є кілька способів задавання та модифікації шляхів до файлів.

Використання модуля *os.path*. Модуль *os.path* містить багато корисних функцій для роботи зі шляхами до файлів. Наприклад, *join()* дає змогу об’єднувати шляхи, *dirname()* повертає каталог, у якому міститься файл, а *basename()* – ім’я файлу зі шляху.

```
import os

# Задання шляху до файлу
file_path = "/path/to/file.txt"

# Модифікація шляху
directory = os.path.dirname(file_path)
filename = os.path.basename(file_path)
new_file_path = os.path.join(directory, "new_" + filename)

print("Початковий шлях:", file_path)
print("Модифікований шлях:", new_file_path)
```

Використання методів рядків. Можна використовувати методи рядків, такі як *split()* та *join()*, для розбиття та об'єднання шляхів.

```
# Задання шляху до файлу
file_path = "/path/to/file.txt"

# Розбивання шляху
parts = file_path.split("/")
parts[-1] = "new_" + parts[-1]
new_file_path = "/".join(parts)

print("Початковий шлях:", file_path)
print("Модифікований шлях:", new_file_path)
```

Використання бібліотеки *pathlib* (з Python 3.4 та вище). Модуль *pathlib* надає об'єктно-орієнтований інтерфейс для роботиі зі шляхами.

```
from pathlib import Path

# Задання шляху до файлу
file_path = Path("/path/to/file.txt")

# Модифікація шляху
new_file_path = file_path.parent / ("new_" + file_path.name)

print("Початковий шлях:", file_path)
print("Модифікований шлях:", new_file_path)
```

Кожен з цих підходів має свої переваги, і ви можете обрати той, який найбільше відповідає вашим потребам і стилістиці коду.

7. Встановлення шляху до поточного каталогу. У Python для встановлення поточного каталогу можна використовувати модуль *os*. У модулі *os* є функція *chdir()*, яка дає змогу змінити поточний каталог.

```
import os
# Отримання поточного каталогу
current_directory = os.getcwd()
print("Поточний каталог:", current_directory)

# Встановлення нового поточного каталогу
new_directory = "/path/to/new/directory"
os.chdir(new_directory)

# Перевірка, що поточний каталог був змінений
updated_directory = os.getcwd()
print("Новий поточний каталог:", updated_directory)
```

Цей код спочатку виводить поточний каталог за допомогою функції *os.getcwd()*, а потім встановлює новий поточний каталог за допомогою функції *os.chdir()*. У результаті виводиться новий поточний каталог.

Важливо враховувати, що зміна поточного каталогу впливає на всі подальші операції в програмі, які використовують відносні шляхи.

8. Одержання шляху до файлу, що виконується, за допомогою атрибута *__file__*. У Python, коли ви виконуєте скрипт, ім'я файлу, який ви виконуєте, доступне через атрибут *__file__* модуля. Це абсолютний або відносний шлях до файлу скрипту, що виконується.

```
import os

# Отримання шляху до файлу, що виконується
current_script_path = os.path.abspath(__file__)

print("Шлях до файлу, що виконується:", current_script_path)
```

У цьому прикладі *os.path.abspath(__file__)* використовується для перетворення відносного шляху до файлу, що виконується, в абсолютний шлях. *__file__* містить шлях до поточного скрипту, включаючи ім'я файлу.

Зверніть увагу, що використання атрибута *__file__* працює лише у виконуваних файлах. У випадку інтерпретації коду в режимі інтерактивної оболонки Python або у випадку, коли модуль імпортується, значення *__file__* буде *None*.

9. Права доступу до файлів і каталогів. У Python є декілька способів отримання та керування правами доступу до файлів і каталогів.

Модуль *os*. Модуль *os* містить функції для взаємодії з операційною системою, включаючи отримання та зміну прав доступу до файлів і каталогів. Наприклад, функція *os.chmod()* дає змогу змінювати права доступу.

```
import os

# Отримання поточних прав доступу до файлу
current_permissions = os.stat("file.txt").st_mode

# Зміна прав доступу (наприклад, встановлення режиму доступу тільки для читання)
os.chmod("file.txt", 0o444)
```

Модуль *stat*. Модуль *stat* надає доступ до атрибутів файлу, таких як права доступу.

```
import os
import stat

# Отримання атрибутів файлу
file_stat = os.stat("file.txt")

# Отримання прав доступу
permissions = file_stat.st_mode
```

Командний рядок. Можна використовувати командний рядок (наприклад, за допомогою команди *chmod* в *Unix* або *icacls* в *Windows*) для зміни прав доступу до файлів.

Зверніть увагу, що для зміни прав доступу до файлів може знадобитися виконання програми в режимі адміністратора, залежно від операційної системи та конкретних обставин. Будьте обережні під час зміни прав доступу до файлів і каталогів, оскільки неправильні права можуть призвести до проблем з безпекою або неправильної роботи програми.

10. Визначення прав доступу. У Python для визначення прав доступу до файлів та каталогів використовуються методи модуля *os* та модуля *sta*t.

Модуль *os* містить функції для взаємодії з операційною системою, включаючи отримання інформації про права доступу до файлів.

```
import os

# Отримання поточних прав доступу до файлу
current_permissions = os.stat("file.txt").st_mode

# Перевірка прав доступу (наприклад, наявність права читання)
if current_permissions & 0o400:
    print("Файл доступний для читання")
```

Модуль *stat* надає доступ до атрибутів файлу, таких як права доступу.

```
import os
import stat

# Отримання атрибутів файлу
file_stat = os.stat("file.txt")

# Отримання прав доступу
permissions = file_stat.st_mode

# Перевірка прав доступу (наприклад, наявність права виконання)
if permissions & stat.S_IXUSR:
    print("Файл доступний для виконання")
```

Обидва ці приклади дають змогу перевірити наявність певних прав доступу до файлу, таких як права читання, запису та виконання. Важливо розуміти, що коди прав доступу можуть варіюватись залежно від операційної системи, а також від різних прав користувачів (власник, група, інші).

11. Перенаправлення вводу/виводу. Функція *print( )*. У Python можна перенаправляти ввід та вивід за допомогою стандартних потоків вводу-виводу (*stdin, stdout, stderr*). Основний метод виводу – це функція *print()*.

Перенаправлення виводу в файл. Можна перенаправити вивід програми в файл, використовуючи стандартні потоки.

```
with open("output.txt", "w") as f:
    print("Hello, world!", file=f)
```

Цей код виводить рядок *"Hello, world!"* у файл з іменем *output.txt*.

Перенаправлення вводу з файлу. Можна також перенаправити ввід програми з файлу, вказавши файл як вхідний потік (*stdin*).

```
import sys

with open("input.txt", "r") as f:
    sys.stdin = f
    user_input = input()
    print("User input:", user_input)
```

У цьому прикладі вхідна інформація для функції *input()* буде братися з файлу *input.txt*.

Вивід на екран. Функція *print()* може бути використана для виведення даних на екран.

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

Функція *print()* також підтримує форматування рядків та інші параметри, які дають змогу контролювати вивід.

12. Функції для роботи з каталогами.

У Python для роботи з каталогами існують кілька корисних функцій та методів.

Створення каталогу. Для створення нового каталогу використовується функція *os.mkdir()* або метод *mkdir()* об'єкта типу *Path* з модуля *pathlib*.

```
import os
from pathlib import Path

# Створення каталогу з використанням os.mkdir()
os.mkdir("new_directory")

# Створення каталогу з використанням pathlib
Path("new_directory").mkdir()
```

Перевірка наявності каталогу. Для перевірки, чи існує каталог, можна використати функцію *os.path.exists()* або метод *exists()* об'єкта типу *Path*.

```
import os
from pathlib import Path

# Перевірка наявності каталогу з використанням os.path.exists()
if os.path.exists("existing_directory"):
    print("Каталог існує")

# Перевірка наявності каталогу з використанням pathlib
if Path("existing_directory").exists():
    print("Каталог існує")
```

Отримання списку вмісту каталогу. Для отримання списку файлів та підкаталогів у каталозі використовується функція *os.listdir()* або метод *iterdir()* об'єкта типу *Path*.

```
import os
from pathlib import Path

# Отримання списку вмісту каталогу з використанням os.listdir()
directory_contents = os.listdir("directory_path")

# Отримання списку вмісту каталогу з використанням pathlib
directory_contents = [item.name for item in Path("directory_path").iterdir()]
```

Видалення каталогу. Для видалення каталогу використовується функція *os.rmdir()* або метод *rmdir()* об'єкта типу *Path*.

```
import os
from pathlib import Path

# Видалення каталогу з використанням os.rmdir()
os.rmdir("directory_to_delete")

# Видалення каталогу з використанням pathlib
Path("directory_to_delete").rmdir()
```

Ці функції та методи дають змогу працювати з каталогами в Python, забезпечуючи широкий спектр можливостей для створення, перевірки наявності, отримання вмісту та видалення каталогів.

13. Виключення, які виникають під час виконання операцій з файлами. Під час роботи з файлами в Python можуть виникати різні виключення, особливо під час виконання операцій, таких як читання, запис і робота з каталогами.

FileNotFoundError – це виключення виникає, коли спроба доступу до файлу, який не існує, не вдається. Наприклад, якщо ви спробуєте відкрити файл для читання або запису, який не існує.

```
try:
    with open("nonexistent_file.txt", "r") as f:
        content = f.read()
except FileNotFoundError:
    print("Файл не знайдено.")
```

PermissionError – це виключення виникає, коли у вас немає необхідних прав доступу до виконання певної операції з файлом. Наприклад, спроба записати файл у каталог, до якого у вас немає прав запису.

```
try:
    with open("/root/some_file.txt", "w") as f:
        f.write("Some content")
except PermissionError:
    print("Недостатні права доступу.")
```

IsADirectoryError та *NotADirectoryError* – ці виключення виникають, коли ви спробуєте виконати операції для файлу, але це виявиться каталогом (або навпаки).

```
try:
    with open("some_directory", "r") as f:
        content = f.read()
except IsADirectoryError:
    print("Це каталог, а не файл.")
except NotADirectoryError:
    print("Це файл, а не каталог.")
```

Перелік питань для самоконтролю

1. Поняття про файл і типи файлів.
2. Відкриття файлу та формат функції **open( )**.
3. Абсолютний та відносний шлях до файлу.
4. Модифікатори відкриття файлу.

5. Особливості роботи з механізмом буферизації файлів.

6. Кодування текстових файлів та особливості роботи з різними кодуваннями.

7. Методи для роботи з файлами.

8. Функції для маніпулювання файлами.

9. Перевірка наявності файлу, розміру файлу, часу останнього доступу, часу створення, часу останньої зміни.

10. Збереження об’єктів у файлі.

11. Функції модуля pickle для роботи з файлами.

12. Використання модуля shelve для зберігання даних у файлі за ключем.

13. Перетворення відносного шляху в абсолютний.

14. Можливі задавання шляхів до файлу та їх модифікація.

15. Встановлення шляху до поточного каталогу.

16. Одержання шляху до файлу, що виконується, за допомогою атрибута **__file__**.

17. Права доступу до файлів і каталогів.

18. Перенаправлення вводу/виводу. Функція **print( ).**

19. Функції для роботи з каталогами.

20. Виключення, які виникають під час виконання операцій з файлами.

Тестові питання

1. Яка функція в Python використовується для відкриття файлу:

а) `open_file()`;

б) `file_open()`;

в) `open()`;

г) `file()`.

2. Який режим відкриття файлу в Python дозволяє тільки читання:

а) `'w'`;

б) `'a'`;

в) `'r'`;

г) `'x'`.

3. Який метод використовується для запису рядка в файл:

а) `write()`;

б) `append()`;

в) `save()`;

г) `insert()`.

4. Яка функція дає змогу зчитати весь вміст файлу в рядок:

а) `readline()`;

б) `read()`;

в) `readlines()`;

г) `fetch()`.

5. Як закрити файл після його використання в Python:

а) `file.close()`;

б) `file.end()`;

в) `file.quit()`;

г) `file.terminate()`.

6. Яка функція забезпечує автоматичне закриття файлу після закінчення роботи з ним:

а) `file.open()`;

б) `file.close()`;

в) `with`;

г) `finally`.

7. Який режим відкриття файлу дає змогує додавати дані в кінець файлу без видалення існуючих даних:

а) `'r+'`;

б) `'w+'`;

в) `'a'`;

г) `'x'`.

8. Що повертає метод `readlines()` під час зчитування з файлу:

а) рядок, що містить весь вміст файлу;

б) список рядків з файлу;

в) словник з іменами файлів і їх розмірами;

г) лише перший рядок файлу.

9. Який з наведених режимів створює новий файл, якщо файл не існує, і перезаписує файл, якщо він існує:

а) `'r'`;

б) `'a'`;

в) `'w'`;

г) `'x'`.

10. Що робить метод `seek(offset)` під час роботи з файлом:

а) установлює новий шлях до файлу;

б) змінює позицію вмісту файлу для подальшого читання / запису;

в) закриває файл;

г) змінює режим відкриття файлу.

Рекомендовані джерела:

Основні: 1; 2; 3; 4; 5; 6.

Допоміжні: 1; 2; 3; 4; 5.

Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.

Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть програму, яка відкриває текстовий файл `data.txt`, читає його вміст і виводить його на екран.

2. Напишіть програму, яка створює новий текстовий файл `output.txt` і записує в нього рядок `"Hello, World!"`.

3. Напишіть програму, яка відкриває існуючий файл `append.txt` і додає до нього рядок `"Appended text"` у кінці файлу.

4. Напишіть програму, яка відкриває текстовий файл `lines.txt`, читає його вміст по рядках і виводить кожен рядок окремо.

5. Напишіть програму, яка відкриває текстовий файл `rewrite.txt`, очищує його вміст і записує новий рядок `"New content"`.

6. Напишіть програму, яка має список рядків і записує ці рядки в текстовий файл `list.txt`, де кожен рядок списку записується на новому рядку.

7. Напишіть програму, яка відкриває файл `numbers.txt`, читає числові значення й обчислює їх суму.

8. Напишіть програму, яка створює CSV файл `data.csv` і записує в нього кілька рядків даних, де кожен рядок містить кілька значень, розділених комами.

9. Напишіть програму, яка відкриває CSV файл `data.csv`, читає його вміст і виводить кожен рядок як список значень.

10. Напишіть програму, яка відкриває JSON файл `data.json`, читає його вміст і виводить інформацію у форматі словника Python.

11. Напишіть програму, яка має словник Python і записує його у JSON файл `output.json`.

12. Напишіть програму, яка відкриває текстовий файл `replace.txt`, замінює всі входження слова `"old"` на `"new"` і зберігає зміни в тому самому файлі.

13. Напишіть програму, яка копіює вміст файлу `source.txt` до нового файлу `destination.txt`.

14. Напишіть програму, яка перевіряє існування файлу `delete_me.txt`, а потім видаляє його, якщо він існує.

15. Напишіть програму, яка рекурсивно обробляє всі файли в директорії `documents`, виводячи імена всіх файлів з розширенням `.txt`.

16. Напишіть програму, яка відкриває текстовий файл `stats.txt`, підраховує кількість рядків, слів і символів у файлі і виводить цю інформацію.

Тема 26
Робота з базами даних у Python на прикладі SQLite

26.1. Підключення до SQLite

Підключення до бази даних *SQLite* в Python зазвичай відбувається за допомогою модуля *sqlite3*.

```
import sqlite3

# Підключення до бази даних
connection = sqlite3.connect('example.db')

# Створення курсора для взаємодії з базою даних
cursor = connection.cursor()

# Закриття курсора та з'єднання
cursor.close()
connection.close()
```

У цьому коді:

- *sqlite3.connect()* використовується для підключення до бази даних SQLite. Якщо база даних не існує, вона буде автоматично створена;

- *connection.cursor()* створює курсор, який використовується для виконання SQL-запитів та отримання результатів;

- після взаємодії з базою даних курсор і з′єднання повинні бути закриті за допомогою методів *close()*.

Важливо пам′ятати, що з'єднання з базою даних повинно бути закрите після завершення всіх операцій, щоб уникнути проблем з блокуванням та ресурсами.

26.2. Зіставлення типів SQLite та Python

У *SQLite* та Python існують відповідні типи даних, які можна використовувати під час роботи з базою даних *SQLite* через модуль *sqlite3*.

1. *TEXT* – у Python відповідником для текстових даних є тип *str*.

2. *INTEGER* – для цілих чисел використовується тип *int* у Python.

3. *REAL* – для дійсних чисел використовується тип *float* у Python.

4. *BLOB* – це великий рядок даних, який зазвичай використовується для збереження бінарних об'єктів, таких як зображення або відео. У Python для представлення *BLOB*-даних може використовуватися тип *bytes*.

5. *NULL* – цей тип відповідає значенню *None* в Python.

Під час створення таблиці в базі даних *SQLite* ви можете використовувати ці типи даних, щоб описати структуру таблиці, і передавати відповідні значення, коли ви вставляєте або отримуєте дані з цієї таблиці через Python.

```
import sqlite3

# Підключення до бази даних
connection = sqlite3.connect('example.db')

# Створення курсора
cursor = connection.cursor()

# Створення таблиці з текстовими, цілими та дійсними полями
cursor.execute('CREATE TABLE IF NOT EXISTS test_table (text_col TEXT, int_col INTEGER, real_col REAL)')

# Вставка даних
cursor.execute('INSERT INTO test_table VALUES (?, ?, ?)', ('Example text', 123, 3.14))

# Збереження змін
connection.commit()

# Закриття курсора та з'єднання
cursor.close()
connection.close()
```

26.3. Отримання курсора

Щоб отримати курсор для виконання запитів до бази даних, ви можете використовувати метод *cursor()* об’єкта з’єднання з базою даних.

```
import sqlite3

# Підключення до бази даних (створення нової, якщо не існує)
conn = sqlite3.connect('example.db')
```

```
# Отримання курсора
cursor = conn.cursor()

# Тепер ви можете використовувати курсор для виконання SQL-запитів
# Наприклад, створення таблиці
cursor.execute('''CREATE TABLE IF NOT EXISTS users
             (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)''')

# Збереження змін до бази даних
conn.commit()

# Закриття з'єднання з базою даних
conn.close()
```

У цьому прикладі після отримання курсора (*cursor = conn.cursor()*) ви можете використовувати цей курсор для виконання різних операцій з базою даних, таких як виконання SQL-запитів для створення таблиць, вставки, оновлення або видалення даних.

26.4. Створення таблиці

1. Створення таблиці для зберігання інформації про користувачів:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Створення таблиці для користувачів
cursor.execute('''CREATE TABLE IF NOT EXISTS users
             (id INTEGER PRIMARY KEY, username TEXT, email TEXT)''')

conn.commit()
conn.close()
```

У цьому прикладі ми створюємо таблицю з назвою *users*, яка містить поля *id*, *username* і *email*. Поле *id* встановлено як основний ключ (*PRIMARY KEY*), що гарантує унікальність значень у цьому полі.

2. Створення таблиці для зберігання списку продуктів у магазині:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Створення таблиці для продуктів
cursor.execute('''CREATE TABLE IF NOT EXISTS products
             (id INTEGER PRIMARY KEY, name TEXT, price REAL)''')

conn.commit()
conn.close()
```

У цьому прикладі ми створюємо таблицю з назвою *products*, яка містить поля *id, name і price*. Поле *price* встановлено як тип *REAL* для зберігання дійсних чисел.

3. Створення таблиці для зберігання історії замовлень:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Створення таблиці для замовлень
cursor.execute('''CREATE TABLE IF NOT EXISTS orders
                 (id INTEGER PRIMARY KEY, customer_id INTEGER,
product_id INTEGER, quantity INTEGER)''')

conn.commit()
conn.close()
```

У цьому прикладі ми створюємо таблицю з назвою *orders*, яка містить поля *id*, *customer_id*, *product_id* і *quantity*. Поля *customer_id* і *product_id* використовуються для зв'язку з таблицями користувачів та продуктів відповідно.

4. Створення таблиці для зберігання контактної інформації:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Створення таблиці для контактної інформації
cursor.execute('''CREATE TABLE IF NOT EXISTS contacts
             (id INTEGER PRIMARY KEY, first_name TEXT, last_name
TEXT, phone TEXT, email TEXT)''')

conn.commit()
conn.close()
```

У цьому прикладі ми створюємо таблицю з назвою contacts, яка містить поля *id, first_name, last_name, phone* і *email*, для зберігання контактної інформації про осіб.

26.5. Додавання даних

1. Додавання даних через *SQL*-запит із рядком запиту:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Додавання даних через SQL-запит зі строкою запиту
cursor.execute("INSERT INTO users (username, email) VALUES
('john_doe', 'john@example.com')")

conn.commit()
conn.close()
```

У цьому прикладі ми вставляємо новий рядок у таблицю *users* зі значеннями *'john_doe'* для поля *username* і *'john@example.com'* для поля *email*.

2. Додавання даних за допомогою параметрів запиту:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Додавання даних за допомогою параметрів запиту
data = ('jane_doe', 'jane@example.com')
cursor.execute("INSERT INTO users (username, email) VALUES (?, ?)", data)

conn.commit()
conn.close()
```

У цьому прикладі ми використовуємо параметри запиту (`?`) та передаємо значення через кортеж *data*. Це робить код більш безпечним щодо *SQL*-ін'єкцій[5].

3. Додавання даних за допомогою списку кортежів:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()
```

[5] SQL-ін’єкції (SQL Injection) – це одна з найпоширеніших уразливостей веб-додатків, яка дає змогу зловмисникам втручатися в запити до бази даних через неналежно оброблені вхідні дані. Це може призвести до викрадення, зміни або видалення даних, а іноді і до повного контролю над сервером бази даних. SQL-ін’єкція виникає, коли зловмисник може вставити або «ін’єктувати» шкідливий SQL-код у запит, що відправляється до бази даних. Це може дати можливість зловмиснику маніпулювати запитами або отримати доступ до конфіденційної інформації.

```
# Додавання даних за допомогою списку кортежів
data = [
    ('alice', 'alice@example.com'),
    ('bob', 'bob@example.com'),
    ('charlie', 'charlie@example.com')
]
cursor.executemany("INSERT INTO users (username, email) VALUES (?, ?)", data)

conn.commit()
conn.close()
```

У цьому прикладі ми використовуємо метод *executemany*, щоб додати багато рядків одночасно. Вхідні дані представлені як список кортежів *data*.

4. Використання іменованих параметрів запиту:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Додавання даних з використанням іменованих параметрів
data = {'username': 'david', 'email': 'david@example.com'}
cursor.execute("INSERT INTO users (username, email) VALUES (:username, :email)", data)

conn.commit()
conn.close()
```

У цьому прикладі ми використовуємо іменовані параметри (*:username*, *:email*) зі словника *data* для передачі даних у запиті.

5. Вставка даних за допомогою об’єктів класу:

```
import sqlite3

class User:
    def __init__(self, username, email):
        self.username = username
        self.email = email

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Створення об'єкта користувача
user = User('emma', 'emma@example.com')

# Додавання даних через об'єкт класу
cursor.execute("INSERT INTO users (username, email) VALUES (?, ?)", (user.username, user.email))

conn.commit()
conn.close()
```

У цьому прикладі ми використовуємо об’єкт класу *User* для представлення даних користувача. Дані об’єкта передаються безпосередньо в *SQL*-запит.

6. Вставка даних з використанням форматування рядка:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Додавання даних з використанням форматування рядка
username = 'frank'
email = 'frank@example.com'
cursor.execute("INSERT INTO users (username, email) VALUES ('{}', '{}')".format(username, email))
```

```
conn.commit()
conn.close()
```

У цьому прикладі ми використовуємо форматування рядка для побудови *SQL*-запиту. Цей підхід менш безпечний порівняно з використанням параметрів запиту і може призвести до *SQL*-ін'єкцій, тому рекомендовано уникати його.

26.6. Встановлення параметрів

1. Встановлення рівня ізоляції транзакцій:

```
import sqlite3

conn = sqlite3.connect('example.db', isolation_level=None)
```

Рівень ізоляції транзакцій визначає ступінь видимості змін, зроблених однією транзакцією, іншим транзакціям, які відбуваються одночасно. Параметр *isolation_level* дає змогу встановити цей рівень. У цьому прикладі значення *None* означає, що база даних використовує рівень ізоляції, який встановлюється за замовчуванням у системі.

2. Встановлення таймауту для очікування блокування:

```
import sqlite3

conn = sqlite3.connect('example.db', timeout=10)
```

Таймаут визначає максимальний час (у секундах), протягом якого база даних буде чекати на блокування. Якщо блокування не може бути отримано протягом цього періоду, генерується виключення *sqlite3.OperationalError*.

3. Встановлення параметра перевірки того самого потоку:

```
import sqlite3

conn = sqlite3.connect('example.db', check_same_thread=False)
```

Цей параметр визначає, чи повинні всі операції з базою даних відбуватися в тому самому потоці, що й створення з'єднання. За замовчуванням, у *SQLite* перевірка виконується, і якщо спроба виконання операцій відбувається з іншого потоку, отримаємо виключення *sqlite3.ProgrammingError*.

4. Встановлення кешування пам'яті:

```
import sqlite3

conn = sqlite3.connect('example.db', cached_statements=100)
```

Цей параметр визначає максимальну кількість скомпільованих *SQL*-запитів, які можуть бути збережені в пам'яті для майбутнього використання. Це може підвищити продуктивність у деяких випадках, особливо якщо виконуються однакові запити багато разів.

5. Встановлення шляху до бібліотеки *SQLite*:

```
import sqlite3

conn = sqlite3.connect('example.db', uri=True)
```

Цей параметр указує, що рядок підключення до бази даних є уніфікованим ідентифікатором ресурсу (*URI*). Це дає змогу використовувати *URI*-схеми, такі як *file:* для вказівки шляху до файлу бази даних, або *:memory:* для створення бази даних в оперативній пам'яті.

26.7. Отримання даних

1. Використання методу *fetchone()* для отримання одного рядка:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()
```

```
# Вибірка першого рядка з таблиці
cursor.execute("SELECT * FROM users WHERE id = 1")
row = cursor.fetchone()

print(row)

conn.close()
```

Метод *fetchone()* використовується для отримання наступного рядка результатів запиту. У цьому прикладі вибираємо перший рядок з таблиці *users*, де *id* дорівнює `1`.

2. Використання методу *fetchall()* для отримання всіх рядків:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка всіх рядків з таблиці
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()

for row in rows:
    print(row)

conn.close()
```

Метод *fetchall()* використовується для отримання всіх рядків результатів запиту одним викликом. У цьому прикладі вибираємо всі рядки з таблиці *users* і виводимо їх.

3. Використання циклу *for* разом з методом *fetchall()* для ітерації за результатами:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка всіх рядків з таблиці та ітерація щодо них
cursor.execute("SELECT * FROM users")
for row in cursor.fetchall():
    print(row)

conn.close()
```

Цей підхід об’єднує вибірку всіх рядків з ітерацією по них у циклі *for*. Це зручно, коли ви хочете обробляти результати запиту одразу.

4. Використання методу *fetchmany()* для отримання обмеженої кількості рядків:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка обмеженої кількості рядків з таблиці
cursor.execute("SELECT * FROM users")
rows = cursor.fetchmany(3)

for row in rows:
    print(row)

conn.close()
```

Метод *fetchmany(n)* використовується для отримання обмеженої кількості рядків результатів запиту, де *n* – кількість рядків, які ви хочете отримати. У цьому прикладі ми вибираємо перші три рядки з таблиці *users*.

5. Використання параметрів запиту для отримання відфільтрованих даних:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка рядків з умовою WHERE
cursor.execute("SELECT * FROM users WHERE age > ?", (18,))
rows = cursor.fetchall()

for row in rows:
    print(row)

conn.close()
```

У цьому прикладі ми використовуємо параметр запиту (`?`) для передачі значення умови *WHERE*. Це дає змогу нам фільтрувати дані на підставі конкретного критерію, у цьому випадку – вибір усіх рядків, де значення поля *age* більше `18`.

6. Використання *ORDER BY* для сортування результатів:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()
```

```
# Вибірка рядків з сортуванням за полем age у спадаючому порядку
cursor.execute("SELECT * FROM users ORDER BY age DESC")
rows = cursor.fetchall()

for row in rows:
    print(row)

conn.close()
```

Запит *ORDER BY* використовується для сортування результатів за певним полем. У цьому прикладі ми сортуємо рядки з таблиці *users* за полем *age* у спадаючому порядку.

7. Використання агрегатних функцій для отримання підсумкових значень:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка середнього віку користувачів
cursor.execute("SELECT AVG(age) FROM users")
average_age = cursor.fetchone()[0]

print("Average age:", average_age)

conn.close()
```

У цьому прикладі використовується агрегатна функція *AVG()* для обчислення середнього віку користувачів у таблиці *users*. Результатом буде одне число – середнє значення віку.

26.8. Оновлення даних

1. Оновлення одного поля у всіх рядках, які відповідають умові:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Оновлення значення поля age для всіх користувачів старше 30 років
cursor.execute("UPDATE users SET age = age + 1 WHERE age > 30")

conn.commit()
conn.close()
```

У цьому прикладі ми оновлюємо значення поля *age* для всіх рядків у таблиці *users*, де вік користувача більше 30 років. Вираз *age = age + 1* збільшує вік кожного користувача на один рік.

2. Оновлення значень у зазначених полях для вибраних рядків:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Оновлення значень полів username та email для користувача з id = 1
cursor.execute("UPDATE users SET username = ?, email = ? WHERE id = ?", ('new_username', 'new_email@example.com', 1))

conn.commit()
conn.close()
```

У цьому прикладі ми оновлюємо значення полів *username* та *email* для користувача з *id = 1*. Замість конкретних значень для цих полів ми використовуємо плейсхолдери, які потім заповнюються викликом *execute()*.

3. Оновлення декількох рядків одночасно за допомогою списку кортежів:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Оновлення значення поля status для кількох користувачів одночасно
data = [
    ('active', 1),
    ('inactive', 2),
    ('active', 3)
]
cursor.executemany("UPDATE users SET status = ? WHERE id = ?", data)

conn.commit()
conn.close()
```

У цьому прикладі ми оновлюємо значення поля *status* для декількох користувачів одночасно. Дані передаються у вигляді списку кортежів, де кожен кортеж містить нове значення поля *status* та *id* користувача.

4. Оновлення значення поля на основі іншого поля:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()
```

```
# Оновлення значення поля full_name на основі об'єднання значень полів first_name та last_name
cursor.execute("UPDATE users SET full_name = first_name || ' ' || last_name")

conn.commit()
conn.close()
```

У цьому прикладі ми оновлюємо значення поля *full_name* для всіх користувачів на основі об’єднання значень полів *first_name* та *last_name*. Оператор `||` використовується для об’єднання рядків у *SQLite*.

5. Оновлення значення поля з використанням підзапиту:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Оновлення значення поля role_id на основі результату під-запиту
cursor.execute("UPDATE users SET role_id = (SELECT id FROM roles WHERE name = 'admin') WHERE id = 1")

conn.commit()
conn.close()
```

У цьому прикладі ми оновлюємо значення поля *role_id* для користувача з *id=1* на основі результату підзапиту, який вибирає *id* ролі з таблиці *roles*, де ім’я ролі дорівнює *'admin'*.

6. Оновлення даних за допомогою умови *CASE WHEN*:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Оновлення значення поля status на основі значення поля age
cursor.execute("UPDATE users SET status = CASE WHEN age >= 18 THEN 'adult' ELSE 'child' END")

conn.commit()
conn.close()
```

У цьому прикладі ми оновлюємо значення поля *status* для всіх користувачів на основі значення поля *age*. Якщо вік користувача більше або дорівнює 18, то значення поля *status* буде *'adult'*, в іншому випадку – *'child'*.

7. Оновлення значення поля з використанням підготовленого запиту:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Оновлення значення поля email за допомогою підготовленого запиту
new_email = 'new_email@example.com'
cursor.execute("UPDATE users SET email = ? WHERE id = ?", (new_email, 1))

conn.commit()
conn.close()
```

26.9. Видалення даних

1. Видалення рядка за умовою:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Видалення користувачів з віком менше 25 років
cursor.execute("DELETE FROM users WHERE age < 25")

conn.commit()
conn.close()
```

У цьому прикладі видаляються всі користувачі з таблиці *users*, у яких вік менше 25 років.

2. Видалення усіх рядків з таблиці:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Видалення всіх рядків з таблиці users
cursor.execute("DELETE FROM users")

conn.commit()
conn.close()
```

У цьому прикладі ми видаляємо всі рядки з таблиці *users*. Обережно використовуйте цей запит, оскільки він видалить всі дані з таблиці без можливості відновлення.

3. Видалення рядків на основі результатів підзапиту:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Видалення користувачів, які мають роль 'guest'
cursor.execute("DELETE FROM users WHERE id IN (SELECT user_id FROM user_roles WHERE role_id = (SELECT id FROM roles WHERE name = 'guest'))")

conn.commit()
conn.close()
```

У цьому прикладі ми видаляємо всіх користувачів, які мають значення *guest*. Ми використовуємо підзапит для отримання *id* значень *guest* з таблиці *roles*, потім знаходимо *user_id* користувачів з цією роллю в таблиці *user_roles* і видаляємо цих користувачів з таблиці *users*.

4. Видалення декількох рядків одночасно за допомогою списку значень:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Видалення користувачів з іменами 'Alice' та 'Bob'
names_to_delete = ('Alice', 'Bob')
cursor.execute("DELETE FROM users WHERE name IN (?, ?)", names_to_delete)

conn.commit()
conn.close()
```

У цьому прикладі ми видаляємо всіх користувачів з іменами *Alice* та *Bob* з таблиці *users*. Ми передаємо імена, які потрібно видалити, як список значень через плейсхолдери запиту.

5. Видалення даних на основі умови з використанням підготовленого запиту:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Видалення користувача з певним id
user_id_to_delete = 1
cursor.execute("DELETE FROM users WHERE id = ?",
(user_id_to_delete,))

conn.commit()
conn.close()
```

У цьому прикладі ми видаляємо користувача з певним *id* з таблиці *users*. Замість жорсткого введення *id* ми використовуємо підготовлений запит з плейсхолдером, щоб передати значення *id* у безпечний спосіб.

26.10. Множинна вставка

У *SQLite* можна використовувати множинну вставку даних для оптимізації швидкодії додавання великої кількості даних у таблицю.

```
import sqlite3

# З'єднання з базою даних
conn = sqlite3.connect('example.db')
cursor = conn.cursor()
```

```
# Дані для вставки
data = [
    ('Alice', 30),
    ('Bob', 25),
    ('Charlie', 35)
]
# Використання масової вставки даних
cursor.executemany("INSERT INTO users (name, age) VALUES (?, ?)", data)

# Підтвердження та закриття з'єднання
conn.commit()
conn.close()
```

У цьому прикладі використовують метод *executemany()*, який виконує однаковий *SQL*-запит для кожного елемента в списку *data*. Кожен елемент *data* – це кортеж з даними для вставки в таблицю.

Цей підхід ефективний, коли Вам потрібно вставити велику кількість рядків даних, оскільки він зменшує кількість обміну даними між Python і *SQLite*, що пришвидшує процес вставки.

26.11. Основні операції з даними в SQLite

1. Створення бази даних:

```
import sqlite3

# Підключення до бази даних (створення, якщо не існує)
conn = sqlite3.connect('example.db')
conn.close()
```

2. Створення таблиці:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Створення таблиці users з полями id, name, age
cursor.execute('''CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY,
                name TEXT,
                age INTEGER
               )''')

conn.commit()
conn.close()
```

3. Вставка даних:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вставка нового користувача
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ('Alice', 30))

conn.commit()
conn.close()
```

4. Вибірка даних:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка всіх користувачів
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()

for row in rows:
    print(row)

conn.close()
```

5. Оновлення даних:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Оновлення віку користувача з ім'ям Alice
cursor.execute("UPDATE users SET age = ? WHERE name = ?", (35, 'Alice'))

conn.commit()
conn.close()
```

6. Видалення даних:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()
```

```
# Видалення користувачів з віком менше 25 років
cursor.execute("DELETE FROM users WHERE age < ?", (25,))

conn.commit()
conn.close()
```

7. Фільтрація даних:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка користувачів з віком менше 30 років
cursor.execute("SELECT * FROM users WHERE age < ?", (30,))
rows = cursor.fetchall()

for row in rows:
    print(row)

conn.close()
```

8. Сортування даних:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка всіх користувачів, відсортованих за зростанням віку
cursor.execute("SELECT * FROM users ORDER BY age")
rows = cursor.fetchall()

for row in rows:
    print(row)

conn.close()
```

9. Групування даних:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка кількості користувачів за кожним віком
cursor.execute("SELECT age, COUNT(*) FROM users GROUP BY age")
rows = cursor.fetchall()

for row in rows:
    print(row)

conn.close()
```

10. Об’єднання даних з різних таблиць:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка імен користувачів та їх ролей
cursor.execute("SELECT users.name, roles.name FROM users INNER
JOIN roles ON users.role_id = roles.id")
rows = cursor.fetchall()

for row in rows:
    print(row)

conn.close()
```

11. Використання підзапитів:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка користувачів з певною роллю
cursor.execute("SELECT * FROM users WHERE id IN (SELECT user_id
FROM user_roles WHERE role_id = 1)")
rows = cursor.fetchall()

for row in rows:
    print(row)

conn.close()
```

12. Використання агрегатних функцій:

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Вибірка середнього віку користувачів
cursor.execute("SELECT AVG(age) FROM users")
average_age = cursor.fetchone()[0]

print("Average age:", average_age)

conn.close()
```

Ці приклади демонструють основні операції з даними в *SQLite*, такі як створення, вставка, вибірка, оновлення та видалення даних, а також фільтрація, сортування, групування, об’єднання даних та використання підзапитів та агрегатних функцій.

Для перевірки наявності таблиці в базі даних *SQLite* можна використати запит *SQL*, який вибере інформацію про таблиці з системної таблиці *sqlite_master*. Ось як це можна зробити за допомогою Python і бібліотеки *sqlite3*:

```
import sqlite3

def table_exists(table_name):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()

    # Виконуємо запит SQL для перевірки існування таблиці
    cursor.execute("SELECT name FROM sqlite_master WHERE type='table' AND name=?", (table_name,))
    result = cursor.fetchone()

    conn.close()

    # Якщо результат не є порожнім, то таблиця існує
    return result is not None

# Перевірка існування таблиці "users"
if table_exists('users'):
    print("Таблиця 'users' існує.")
else:
    print("Таблиця 'users' не існує.")
```

У цьому коді функція *table_exists()* приймає назву таблиці як аргумент і виконує запит до системної таблиці *sqlite_master* для перевірки, чи є таблиця з такою назвою в базі даних. Результат виводиться у вигляді повідомлення, яке показує, чи існує вказана таблиця.

Щоб запобігти *SQL*-ін'єкціям, краще використовувати параметризовані запити. Вони дають змогу передавати параметри окремо від запиту *SQL*, що робить неможливим виконання шкідливого коду як *SQL*-ін'єкцію.

У Python і бібліотеці *sqlite3* для виконання параметризованих запитів можна використовувати метод *execute()* разом із запитом *SQL*, у якому містяться плейсхолдери для параметрів. Параметри передаються у вигляді кортежу або списку як другий аргумент методу *execute()*.

```
import sqlite3

# Параметри запиту
name = "Alice"
age = 30

# Підключення до бази даних
conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Виконання параметризованого запиту
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", (name, age))

# Підтвердження та закриття з'єднання
conn.commit()
conn.close()
```

У цьому прикладі плейсхолдери `?` використовуються в запиті *SQL* для позначення місця, куди будуть вставлені параметри. Потім параметри (*name, age*) передаються як другий аргумент методу *execute()*. Використання параметризованих запитів дає змогу безпечно передавати дані до *SQL*-запитів і запобігає *SQL*-ін'єкціям.

SQL-ін'єкція – це вид атаки на систему, використовуючи *SQL*-запити, які неправильно обробляються або не перевіряються на вміст шкідливого коду, що може бути введений користувачем через інтерфейс введення даних, такий як форма вебсайту. Основна ідея полягає в тому, що зловмисник вставляє спеціально сформовані *SQL*-запити або фрагменти коду *SQL* у механізм обробки запитів, що призводить до небажаних наслідків.

Деякі способи, які можуть використовуватися для *SQL*-ін'єкцій:

1. Введення даних з форми. Зловмисник може вставити *SQL*-код у веб-форму, яка потім буде використовуватися для формування *SQL*-запиту без належної перевірки або екранування введених даних.

2. *URL*-параметри. *SQL*-ін'єкція може відбуватися через *URL*-параметри, якщо параметри *URL* безпосередньо використовуються для формування *SQL*-запиту.

3. *Cookies*. Якщо дані з *cookies* безпосередньо використовуються для формування *SQL*-запиту, то вони можуть бути використані для *SQL*-ін'єкцій.

4. Запити *HTTP*. *SQL*-ін'єкція може відбуватися через *HTTP*-запити, які передаються серверу і потім використовуються для формування *SQL*-запитів.

Наприклад, якщо веб-додаток має форму для входу, зловмисник може ввести спеціально сформований рядок у поле логіну або паролю, який містить *SQL*-код. Якщо ці дані безпосередньо використовуються для формування *SQL*-запиту (наприклад, для перевірки логіна та пароля в базі даних), то зловмисник може отримати доступ до конфіденційної інформації або навіть виконати несанкціоновані дії в базі даних.

Перелік питань для самоконтролю

1. Підключення до *SQLite*.
2. Зіставлення типів *SQLite* та Python.
3. Отримання курсору.
4. Створення таблиці.
5. Додавання даних.
6. Встановлення параметрів.
7. Отримання даних.
8. Оновлення даних.
9. Видалення даних.
10. Виконання запитів до бази даних.
11. Множинна вставка.
12. Основні операції з даними в *SQLite*.

Тестові питання

1. Що таке SQLite:

а) реляційна база даних, що використовує мову SQL;

б) база даних, що працює на основі об’єктів Python;

в) система керування базами даних, що не вимагає сервера;

г) мова програмування для роботи з базами даних.

2. Яка бібліотека використовується для роботи з SQLite в Python:

а) psycopg2;

б) SQLAlchemy;

в) sqlite3;

г) MySQLdb.

3. Яка функція в модулі `sqlite3` використовується для підключення до бази даних SQLite:

а) connect();

б) open();

в) create_connection();

г) init_database().

4. Що таке курсор у контексті роботи з базою даних SQLite:

а) запит, який повертає результати запиту до бази даних;

б) з’єднання між Python та SQLite базою даних;

в) об’єкт, що використовується для виконання запитів до бази даних та отримання результатів;

г) метод для виконання структурних змін у базі даних.

5. Яка функція використовується для виконання запитів SELECT до бази даних SQLite:

а) execute();

б) select();

в) fetchone();

г) query().

6. Як можна передати параметри запиту SQL для запобігання SQL-ін'єкцій у функцію `execute()`:

а) зазначити параметри у форматі `(?, ?)` у рядку запиту та передати значення в кортежі;

б) зазначити параметри у форматі `{}`, а потім використати метод `format()` для вставки значень;

в) використати оператор `%` для вставки значень у рядок запиту;

г) немає можливості передачі параметрів у функцію `execute()`.

7. Як отримати всі результати запиту SELECT у Python:

а) викликати метод `fetchall()` курсора;

б) викликати метод `fetchone()` курсора в циклі до отримання `None`;

в) використовувати функцію `execute()` без передачі параметрів;

г) метод `fetchall()` не існує; результати автоматично повертаються під час виконання запиту.

8. Яка функція використовується для збереження змін у базі даних після виконання запиту INSERT, UPDATE або DELETE:

а) save();

б) commit();

в) store();

г) push().

9. Як перевірити, чи існує таблиця в базі даних SQLite:

а) виконати запит типу `SELECT * FROM table_name`;

б) викликати метод `table_exists()` курсора;

в) виконати запит `PRAGMA table_info(table_name)`;

г) викликати метод `table_exists()` модуля `sqlite3`.

10. Як видалити таблицю з бази даних SQLite:

а) виконати запит типу `DROP TABLE table_name`;

б) викликати метод `delete_table()` курсора;

в) виконати запит `DELETE FROM table_name`;

г) викликати метод `drop_table()` модуля `sqlite3`.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси інтернет: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть програму, яка підключається до бази даних SQLite з іменем `example.db`. Якщо база даних не існує, вона повинна бути створена.

2. Напишіть програму, яка створює таблицю `users` в базі даних `example.db`. Таблиця повинна містити стовпці для `id`, `name` і `email`.

3. Напишіть програму, яка вставляє новий запис у таблицю `users` з іменем `"Alice"` та електронною поштою `"alice@example.com"`.

4. Напишіть програму, яка вибирає всі записи з таблиці `users` і виводить їх на екран.

5. Напишіть програму, яка оновлює електронну пошту користувача з ім’ям `"Alice"` на `"alice@newdomain.com"`.

6. Напишіть програму, яка видаляє користувача з таблиці `users` з ім’ям `"Alice"`.

7. Напишіть програму, яка вибирає і виводить всі записи з таблиці `users`, де електронна пошта містить `"example.com"`.

8. Напишіть програму, яка додає індекс до стовпця `email` у таблиці `users`.

9. Напишіть програму, яка підраховує і виводить кількість записів у таблиці `users`.

10. Напишіть програму, яка використовує транзакцію для вставки двох нових записів у таблицю `users`. Якщо вставка одного з записів не вдається, транзакція повинна бути скасована.

11. Напишіть програму, яка створює дві таблиці: `authors` і `books`. Таблиця `books` повинна містити зовнішній ключ, який посилається на таблицю `authors`.

12. Напишіть програму, яка виконує SQL-запит для вибірки всіх книг разом з ім’ям автора з таблиць `books` і `authors`.

13. Напишіть програму, яка вибирає всі записи з таблиці `users` і сортує їх за іменем у порядку зростання.

14. Напишіть програму, яка вибирає всі записи з таблиці `users`, де ім’я починається з букви `"J"`.

15. Напишіть програму, яка вибирає всі унікальні електронні пошти з таблиці `users`.

16. Напишіть програму, яка використовує параметри для вибірки записів з таблиці `users`, де ім’я користувача дорівнює заданому значенню.

17. Напишіть програму, яка створює таблицю `products`, де стовпець `id` автоматично інкрементується під час додавання нового запису.

18. Напишіть програму, яка видаляє таблицю `users` з бази даних `example.db`.

19. Напишіть програму, яка читає дані з CSV-файлу `data.csv` і вставляє ці дані в таблицю `products`.

20. Напишіть програму, яка створює резервну копію бази даних `example.db` у файл `backup.db` і потім відновлює базу даних з цієї резервної копії.

Тема 27
Робота з датами та часом у Python

27.1. Модуль datetime

Модуль *datetime* – це вбудований модуль мови програмування Python, який надає зручні засоби для роботи з датами та часом. Цей модуль дає змогу створювати, обробляти та форматувати дати та час, виконувати арифметичні операції з ними, а також працювати з часовими зонами.

Основні класи та функції модуля *datetime* включають:

1. *datetime* – клас, який представляє об'єкти дати та часу. Об'єкти цього класу мають атрибути, такі як рік, місяць, день, година, хвилина, секунда та мікросекунда;

2. *date* – клас, який представляє об'єкти дати без часу;

3. *time* – клас, який представляє об'єкти часу без дати;

4. *timedelta* – клас, який представляє різницю між двома об'єктами *datetime*, *date* або *time*;

5. *timezone* – клас, який представляє часовий пояс;

6. *datetime.now()* – функція, яка повертає об'єкт *datetime*, який представляє поточну дату та час;

7. *datetime.strptime()* – функція, яка конвертує рядок у об'єкт *datetime* за допомогою заданого формату;

8. *datetime.strftime()* – метод об'єкта *datetime*, який перетворює об'єкт *datetime* у рядок за допомогою заданого формату;

9. *datetime.timedelta()* – функція для створення об'єкта *timedelta* з заданою кількістю днів, секунд та мікросекунд;

10. *datetime.replace()* – метод об'єкта *datetime*, який повертає копію об'єкта *datetime* зі зміненими атрибутами (рік, місяць, день, година, хвилина, секунда та мікросекунда).

Наприклад, можна створити об'єкт *datetime*, який представляє поточну дату та час, за допомогою функції *datetime.now()*:

```
import datetime

current_datetime = datetime.datetime.now()
print(current_datetime)
```

Модуль *datetime* надає багато інших функцій та можливостей для роботи з датами та часом у Python, які дають змогу легко й ефективно виконувати різноманітні завдання, пов'язані з обробкою дат.

27.2. Клас date

Клас *date* є одним із класів, доступних у модулі *datetime* мови програмування Python. Він призначений для представлення дати без часової інформації.

Основні характеристики та методи класу *date* включають:

1. *date(year, month, day)* – конструктор класу *date*, який створює об'єкт *date* з заданими значеннями року, місяця та дня;

2. об'єкт *date* має три атрибути: *year, month* та *day*, які представляють відповідно рік, місяць та день дати;

3. *date.today()* – статичний метод класу *date*, який повертає об'єкт *date*, який представляє поточну дату;

4. *date.weekday()* – метод, який повертає номер дня тижня (понеділок – 0, вівторок – 1, ..., неділя – 6) для заданої дати;

5. *date.isoweekday()* – метод, який повертає номер дня тижня за *ISO* (понеділок – 1, вівторок – 2, ..., неділя – 7) для заданої дати;

6. *date.replace(year=None, month=None, day=None)* – метод, який повертає копію об’єкта *date* зі зміненими атрибутами. Якщо значення не вказано, вони залишаються такими, які є;

7. *date.strftime(format)* – метод, який перетворює об’єкт *date* у рядок за допомогою заданого формату;

8. *date.__str__()* – метод, який повертає рядок, що представляє об’єкт *date*, у форматі *"YYYY-MM-DD"*.

Наприклад, можна створити об’єкт *date* для представлення 15 січня 2024 року та виконати деякі операції з ним:

```
import datetime

# Створення об'єкта date для 15 січня 2024 року
d = datetime.date(2024, 1, 15)

# Виведення атрибутів дати
print("Year:", d.year)
print("Month:", d.month)
print("Day:", d.day)

# Повернення номера дня тижня (понеділок - 0, ..., неділя - 6)
print("Weekday:", d.weekday())

# Повернення номера дня тижня за ISO (понеділок - 1, ..., неділя - 7)
print("ISO Weekday:", d.isoweekday())

# Перетворення об'єкта date у рядок
print("String representation:", d.strftime("%Y-%m-%d"))
```

Ці методи та атрибути дають змогу працювати з об’єктами *date* для представлення та обробки дат у Python.

27.3. Клас часу

У модулі *datetime* мови програмування Python є клас *time*, який призначений для представлення часу без інформації про дату.

Основні характеристики та методи класу *time* включають:

1. *time(hour, minute, second, microsecond)* – конструктор класу *time*, який створює об’єкт *time* з заданими значеннями годин, хвилин, секунд та мікросекунд;

2. об’єкт *time* має чотири атрибути: *hour, minute, second* та *microsecond*, які представляють відповідно години, хвилини, секунди та мікросекунди;

3. *time.replace(hour=None, minute=None, second=None, microsecond=None)* – метод, який повертає копію об’єкта *time* зі зміненими атрибутами. Якщо значення не вказано, вони залишаються такими, які є;

4. *time.strftime(format)* – метод, який перетворює об’єкт *time* в рядок за допомогою заданого формату;

5. *time.__str__()* – метод, який повертає рядок, що представляє об’єкт *time*, у форматі *"HH:MM:SS.microsecond"*.

Наприклад, можна створити об’єкт *time* для представлення 9 годин, 30 хвилин та 15 секунд:

```
import datetime

# Створення об'єкта time для 9:30:15
t = datetime.time(9, 30, 15)

# Виведення атрибутів часу
print("Hour:", t.hour)
print("Minute:", t.minute)
print("Second:", t.second)
print("Microsecond:", t.microsecond)

# Перетворення об'єкта time у рядок
print("String representation:", t.strftime("%H:%M:%S.%f"))
```

27.4. Клас datetime

Клас *datetime* є одним із основних класів у модулі *datetime* мови програмування Python. Він призначений для представлення об’єктів, які містять інформацію про дату та час.

Основні характеристики та методи класу *datetime* включають:

1. *datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0)* – конструктор класу *datetime*, який створює об’єкт *datetime* з заданими значеннями року, місяця, дня, годин, хвилин, секунд та мікросекунд. Години, хвилини, секунди та мікросекунди є необов’язковими параметрами та за замовчуванням встановлюються як 0;

2. об’єкт *datetime* має сім атрибутів: *year, month, day, hour, minute, second* та *microsecond*, які представляють відповідно рік, місяць, день, години, хвилини, секунди та мікросекунди;

3. *datetime.now()* – статичний метод класу *datetime*, який повертає об’єкт *datetime*, що представляє поточну дату та час;

4. *datetime.strptime(date_string, format)* – статичний метод класу *datetime*, який конвертує рядок у об’єкт *datetime* за допомогою заданого формату;

5. *datetime.strftime(format)* – метод, який перетворює об’єкт *datetime* у рядок за допомогою заданого формату;

6. *datetime.replace(year=None, month=None, day=None, hour=None, minute=None, second=None, microsecond=None)* – метод, який повертає копію об’єкта *datetime* зі зміненими атрибутами. Якщо значення не вказано, вони залишаються такими, які є;

7. *datetime.__str__()* – метод, який повертає рядок, що представляє об’єкт *datetime*, у форматі *"YYYY-MM-DD HH:MM:SS.microsecond"*.

Наприклад, можна створити об’єкт datetime для представлення 1 січня 2024 року о 9:30 ранку та вивести деякі його характеристики:

```
import datetime

# Створення об'єкта datetime для 1 січня 2024 року о 9:30 ранку
dt = datetime.datetime(2024, 1, 1, 9, 30)

# Виведення атрибутів datetime
print("Year:", dt.year)
print("Month:", dt.month)
print("Day:", dt.day)
print("Hour:", dt.hour)
print("Minute:", dt.minute)

# Перетворення об'єкта datetime у рядок
print("String representation:", dt.strftime("%Y-%m-%d %H:%M:%S"))
```

27.5. Перетворення з рядка на дату

У Python для перетворення рядка в об’єкт дати можна використовувати метод *strptime()* класу *datetime.datetime* (який є статичним методом). Цей метод дає змогу вказати формат рядка дати та часу, який потрібно розпізнати.

```
import datetime

# Рядок, який містить дату та час у форматі "рік-місяць-день година:хвилина:секунда"
date_string = "2024-05-17 14:30:00"

# Перетворення рядка у об'єкт datetime за допомогою strptime()
date_object = datetime.datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S")

print("Результат перетворення:", date_object)
```

У цьому прикладі *%Y* позначає рік у чотиризначному форматі, *%m* – місяць, *%d* – день, *%H* – година у 24-годинному форматі, *%M* – хвилина, а *%S* – секунда. Формат рядка дати та часу має відповідати формату рядка, який передається до методу *strptime()*.

Якщо формат рядка дати не відповідає формату, переданому до *strptime()*, виникне помилка *ValueError*. Тому важливо вказати правильний формат для перетворення рядка в дату.

27.6. Отримання дат і часу

У Python для отримання поточної дати та часу можна використовувати класи *datetime* та *date* з модуля *datetime*. Вони надають методи для цієї мети.

```
import datetime

current_datetime = datetime.datetime.now()
print("Поточна дата та час:", current_datetime)
```

Цей код створить об’єкт *datetime*, який представляє поточну дату та час, і виведе його.

Якщо потрібна лише поточна дата без часу, можна використати клас *date:*

```
current_date = datetime.date.today()
print("Поточна дата:", current_date)
```

Це створить об’єкт *date*, який представляє поточну дату, без часу, і виведе її.

27.7. Складання та віднімання дат і часу

У Python для складання та віднімання дат і часу використовується клас *timedelta* з модуля *datetime*. Цей клас дає змогу виконувати арифметичні операції з об'єктами *datetime*, *date* та *time*.

Складання та віднімання дат і часу з використанням *timedelta*:

```
import datetime

# Поточна дата та час
current_datetime = datetime.datetime.now()

# Створення timedelta
delta = datetime.timedelta(days=5, hours=3, minutes=30)

# Складання дати та часу з timedelta
result_datetime = current_datetime + delta
print("Результат складання:", result_datetime)

# Віднімання timedelta від дати та часу
result_datetime_subtract = current_datetime - delta
print("Результат віднімання:", result_datetime_subtract)
```

У цьому прикладі *delta* є об'єктом *timedelta*, який представляє різницю у 5 днів, 3 години та 30 хвилин. Ми можемо додати цей *timedelta* до поточної дати та часу або відняти його від неї, отримуючи відповідно нові дати та часи.

Потрібно звернути увагу, що *timedelta* може містити різницю у днях, секундах, мікросекундах тощо, що дає змогу точно виконувати арифметичні операції з датами та часом у Python.

27.8. Властивості timedelta

Об’єкти *timedelta* в модулі *datetime* мають ряд корисних властивостей, які дають змогу отримувати різні атрибути та значення часу, що містяться в об’єкті *timedelta*:

1. *days* – кількість днів у *timedelta*;

2. *seconds* – кількість секунд у *timedelta*, за винятком днів;

3. *microseconds* – кількість мікросекунд у *timedelta*, за винятком днів та секунд;

4. *total_seconds()* – повна кількість секунд у *timedelta*, враховуючи всі частини, включаючи дні, секунди та мікросекунди;

5. *days* – кількість днів;

6. *total_days* – повна кількість днів, враховуючи частини днів у форматі дробу.

Наприклад, якщо маємо *timedelta* з різницею у 5 днів, 3 години та 30 хвилин, то можемо отримати різні значення властивостей:

```
import datetime

delta = datetime.timedelta(days=5, hours=3, minutes=30)

print("Кількість днів:", delta.days)
print("Кількість секунд:", delta.seconds)
print("Кількість мікросекунд:", delta.microseconds)
print("Повна кількість секунд:", delta.total_seconds())
print("Повна кількість днів (з урахуванням годин та хвилин):",
delta.total_days)
```

27.9. Порівняння дат

У Python порівняння дат і часу можна виконати за допомогою стандартних операторів порівняння, таких як `<`, `<=`, `>`, `>=`, `==`, `!=`. Ці оператори працюють з об’єктами *datetime*, *date* та *time*.

```
import datetime

# Створення дати
date1 = datetime.date(2024, 5, 17)
date2 = datetime.date(2024, 5, 18)

# Порівняння дат
if date1 < date2:
    print("date1 менше за date2")
elif date1 == date2:
    print("date1 дорівнює date2")
else:
    print("date1 більше за date2")
```

Аналогічно можна порівнювати об’єкти *datetime* та *time*. Для об’єктів *datetime* порівнюватиметься як дата, так і час, а для об’єктів *time* – тільки час.

Перелік питань для самоконтролю

1. Модуль datetime.
2. Клас date.
3. Клас часу.
4. Клас datetime.
5. Перетворення з рядка на дату.
6. Отримання дат та часу.
7. Складання та віднімання дат і часу.
8. Властивості timedelta.
9. Порівняння дат.

Тестові питання

1. Які типи об’єктів використовуються для роботи з датами та часом у Python:

а) `date` та `time`;

б) `datetime` та `timedelta`;

в) `calendar` та `timezone`;

г) `timestamp` та `duration`.

2. Що таке об’єкт `datetime` в Python:

а) об’єкт, який представляє лише дату;

б) об’єкт, який представляє лише час;

в) об’єкт, який представляє як дату, так і час;

г) об’єкт, який представляє тільки відмітку часу.

3. Яка функція використовується для отримання поточної дати та часу в Python:

а) `today()`;

б) `now()`;

в) `current_datetime()`;

г) `current_time()`.

4. Яка функція перетворює рядок у об’єкт `datetime`:

а) `str_to_datetime()`;

б) `parse_datetime()`;

в) `datetime.parse()`;

г) `datetime.strptime()`.

5. Що таке об’єкт `timedelta` в Python:

а) об’єкт, який представляє різницю між двома датами або часами;

б) об’єкт, який представляє період часу в днях;

в) об’єкт, який представляє тільки час;

г) об’єкт, який представляє різницю між двома точками в часі.

6. Яка функція використовується для додавання або віднімання періоду часу від об'єкта `datetime`:

а) `add_timedelta()`;

б) `subtract_timedelta()`;

в) `add()` та `subtract()`;

г) `timedelta.add()` та `timedelta.subtract()`.

7. Яка функція визначає, чи є рік високосним у Python:

а) `is_leap_year()`;

б) `leap_year()`;

в) `isleap()`;

г) `check_leap_year()`.

8. Яка функція визначає різницю між двома датами або часами в Python:

а) `difference()`;

б) `delta()`;

в) `diff()`;

г) `__sub__()` (віднімання).

9. Яка функція конвертує об'єкт `datetime` у рядок за допомогою певного формату:

а) `to_str()`;

б) `format_datetime()`;

в) `strftime()`;

г) `datetime_to_str()`.

10. Який метод об'єкта `datetime` використовується для зміни компонентів дати чи часу:

а) `change()`;

б) `set()`;

в) `modify()`;

г) `replace()`.

Рекомендовані джерела:
Основні: 1; 2; 3; 4; 5; 6.
Допоміжні: 1; 2; 3; 4; 5.
Інформаційні ресурси мережі «Інтернет»: 1; 2; 3; 4; 5.
Міжнародні видання: 1; 2; 3; 4; 5.

Практичні завдання

1. Напишіть програму, яка виводить поточну дату та час у форматі `YYYY-MM-DD HH:MM:SS`.
2. Напишіть програму, яка приймає дату у форматі `YYYY-MM-DD` і виводить її у форматі `DD/MM/YYYY`.
3. Напишіть програму, яка обчислює і виводить кількість днів між двома датами `2024-01-01` і `2024-08-06`.
4. Напишіть програму, яка приймає дату і кількість днів, додає ці дні до дати і виводить нову дату.
5. Напишіть програму, яка обчислює вік особи на основі дати народження у форматі `YYYY-MM-DD` та поточної дати.
6. Напишіть програму, яка приймає рядок дати у форматі `March 5, 2024` і перетворює його в об’єкт дати Python.
7. Напишіть програму, яка виводить поточну дату у вигляді слова, наприклад, "6 серпня 2024 року".
8. Напишіть програму, яка виводить день тижня для заданої дати у форматі `YYYY-MM-DD`.
9. Напишіть програму, яка порівнює дві дати і виводить, яка з них є більш ранньою.
10. Напишіть програму, яка приймає час у форматі `HH:MM:SS` і виводить його у форматі `HH:MM AM/PM`.
11. Напишіть програму, яка обчислює різницю між двома часами `14:30:00` і `18:45:00` і виводить її у форматі `години:хвилини:секунди`.
12. Напишіть програму, яка вставляє поточний час у рядок повідомлення, наприклад: `"Зараз 14:30:00"`.
13. Напишіть програму, яка приймає дату і час у форматі `YYYY-MM-DD HH:MM:SS` і перетворює їх у UNIX-час.

14. Напишіть програму, яка обчислює кількість робочих днів між двома датами, ігноруючи вихідні.

15. Напишіть програму, яка виводить календар на поточний місяць.

16. Напишіть програму, яка перевіряє, чи є заданий рік високосним.

17. Напишіть програму, яка обчислює тривалість події на основі часу початку і закінчення у форматі `YYYY-MM-DD HH:MM:SS`.

18. Напишіть програму, яка виводить поточний час у форматі `ISO 8601`, наприклад, `2024-08-06T14:30:00`.

19. Напишіть програму, яка конвертує дату і час з однієї часової зони в іншу.

20. Напишіть програму, яка використовує бібліотеку `dateutil` для парсингу та маніпуляції датами, наприклад, розрахунку наступної дати через тиждень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Основний

1. Алхімова С. М. Об’єктно-орієнтоване програмування. Київ : КПІ імені Ігоря Сікорського, 2019. 190 с.

2. Васильєв О. М. Програмування в PYTHON. Теорія і практика : навчальний посібник. Тернопіль : Ліра-К, 2023. 462 с.

3. Васильєв О. М. Програмування мовою Python. Тернопіль : Навчальна книга – Богдан, 2019. 504 с.

4. Гришанович Т. О. Основи об’єктно-орієнтованого програмування. Харків, 2020. 102 с.

5. Костюченко А. О. Основи програмування мовою Python : навч. посіб. Чернігів : ФОП Баликіна С. М., 2020. 180 с.

6. Яковенко А. В. Основи програмування. Python : підручник. Частина 1. Київ : КПІ імені Ігоря Сікорського, 2018. 195 с.

Допоміжний

1. Григорович В. Г. Алгоритмізація та програмування : навчальний посібник. Львів : Магнолія-2006, 2023. 357 с.

2. Григорович В. Г. Алгоритмізація та програмування : навчальний посібник. Львів : Магнолія-2006, 2024. 268 с.

3. Злобін Г. Г. Алгоритмізація та програмування : підручник. Київ : Каравела, 2023. 168 с.

4. Ковалюк Т. В. Алгоритмізація та програмування : підручник. Львів : Магнолія-2006, 2021. 400 с.

5. Карпенко М. Ю., Манакова Н. О., Гавриленко І. О. Технології створення програмних продуктів та інформаційних систем : навч. посібник / Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. Харків : ХНУМГ ім. О. М. Бекетова, 2017. 93 с.

Інформаційні ресурси мережі «Інтернет»

1. The Official Home of the Python Programming Language. URL : https://www.python.org/

2. Документація Python. URL : https://www.python.org/doc/

3. Документація NumPy. URL : https://numpy.org/doc/stable/

4. Підручник з Python. URL : https://docs.python.org/uk/3/tutorial/index.html

5. Tkinter Documentation. URL : https://wiki.python.org/moin/TkInter

Міжнародні видання

1. Matthers E. Python Crash Course. No Starch Press, 2019. 544 p.

2. Browning J. B., Alchin M. Pro Python 3: Features and Tools for Professional Development. 3 rd ed. Apress, 2019. 458 p.

3. Stephenson B. The Python Workbook: A Brief Introduction with Exercises and Solutions. 2 nd ed. Springer, 2019. 219 p.

4. Hunt J. A Beginners Guide to Python 3 Programming. Springer, 2019. 527 p.

5. Hunt J. Advanced Guide to Python 3 Programming. Springer, 2019. 524 p.

Навчальне видання

Лаговський Володимир Вікторович,
Філоненко Михайло Миколайович,
Грищенко Світлана Миколаївна

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

Навчальний посібник

Відповідальний за випуск	*Н. Б. Добрянська*
Редактор	*М. П. Клименко*
Форматування та комп'ютерна верстка	*Д. П. Завальніцька*
Художник обкладинки	*І. С. Хоменко*

Здано до друку 05.12.2025. Формат 29,7 × 41,99
Папір офсетний № 1. Гарнітура «Times New Roman»
Ум. друк. арк. 14.5
Наклад 300 прим. Замовлення № 1128

Підготовлено до друку науково-організаційним відділом
Державного податкового університету
08205, вул. Університетська, 31, м. Ірпінь, Київська область,
Україна

*Свідоцтво про внесення суб'єкта видавничої справи
до державного реєстру видавців, виготовлювачів і
розповсюджувачів видавничої продукції
Серія ДК № 7669 від 20.09.2022*