Міністерство освіти і науки України
Міжнародний гуманітарний університет
Кафедра комп’ютерної інженерії та інноваційних технологій

Педяш В.В.

СКРИПТОВІ МОВИ ПРОГРАМУВАННЯ

Опорний конспект лекцій
для здобувачів вищої освіти,
які навчаються за спеціальностями
галузі «Інформаційні технології»

Одеса 2025

Затверджено Вченою Радою Міжнародного гуманітарного університету.
Протокол № 5 від 20 січня 2025 р.

Педяш В.В. Скриптові мови програмування. Опорний конспект лекцій для здобувачів вищої освіти, які навчаються за спеціальностями галузі «Інформаційні технології» [Електронне видання]. Кафедра комп'ютерної інженерії та інноваційних технологій Міжнародного гуманітарного університету. Одеса, 2025. 34 с.

Дисципліна «Скриптові мови програмування» спрямована на формування у здобувачів вищої освіти спеціальності Інженерія програмного забезпечення компетентностей, пов'язаних із розробкою програмного забезпечення із використанням сучасних скриптових мов програмування. Вона орієнтована на формування здатності розробляти алгоритми, створювати програмні модулі та програмні системи для розв'язання прикладних задач різної складності, застосовувати стандартні та сторонні бібліотеки для обробки даних, автоматизації процесів та інтеграції програмних компонентів.

ЗМІСТ

Лекція 1. Поняття скриптових мов програмування

Скриптові мови програмування є важливою складовою сучасної програмної інженерії, оскільки вони забезпечують швидку розробку програмних компонентів, автоматизацію обчислювальних процесів і ефективну взаємодію з операційним середовищем. На відміну від компільованих мов програмування, скриптові мови орієнтовані на інтерпретацію програмного коду під час виконання, що дає змогу значно скоротити цикл розробки та тестування програмних рішень.

Під скриптовою мовою програмування розуміється мова, призначена для написання програм, які виконуються інтерпретатором без попередньої компіляції у машинний код. Такий підхід забезпечує гнучкість, портативність та зручність внесення змін у програмний код. Скриптові мови широко застосовуються для автоматизації системних задач, обробки даних, розробки вебзастосунків, створення тестових сценаріїв, а також інтеграції різних програмних компонентів.

Однією з ключових характеристик скриптових мов є динамічна типізація. Це означає, що тип змінної визначається під час виконання програми, а не на етапі її компіляції. Такий підхід дає змогу розробнику швидко створювати програмні рішення без необхідності явного оголошення типів даних. Разом із тим це накладає певні обмеження, пов’язані з контролем коректності типів і можливими помилками під час виконання.

Ще однією характерною ознакою є автоматичне керування пам’яттю. У більшості скриптових мов використовується механізм збирання сміття, який автоматично звільняє пам’ять, зайняту об’єктами, що більше не використовуються. Це спрощує розробку програм і зменшує ймовірність виникнення помилок, пов’язаних із неправильним управлінням пам’яттю.

Скриптові мови часто використовуються як вбудовані мови в інших програмних системах. Наприклад, вони можуть застосовуватися для налаштування поведінки програм, написання макросів або реалізації розширень. Це дає змогу відокремити логіку керування від основного програмного коду і забезпечити більшу гнучкість системи.

Серед найбільш поширених скриптових мов можна виділити Python, JavaScript, PHP та Ruby. Кожна з цих мов має свої особливості, але всі вони орієнтовані на швидку розробку, простоту синтаксису та високу продуктивність праці програміста.

Синтаксис скриптових мов визначає правила запису програмного коду, включаючи структуру операторів, способи оголошення змінних, використання виразів та організацію програмних блоків. Синтаксис зазвичай є більш простим і читабельним порівняно з низькорівневими мовами програмування, що робить скриптові мови доступними для широкого кола користувачів.

Основною одиницею будь-якої програми є інструкція або оператор. Оператори можуть виконувати різні дії, такі як присвоєння значень змінним, виконання арифметичних операцій, виклик функцій або керування потоком виконання програми. У скриптових мовах оператори, як правило, записуються у вигляді текстових рядків, що виконуються послідовно.

Змінні у скриптових мовах використовуються для збереження даних під час виконання програми. Вони можуть містити значення різних типів, таких як числа, рядки, списки або об’єкти. Завдяки динамічній типізації змінна може змінювати свій тип у процесі виконання програми, що дає змогу реалізувати гнучку логіку обробки даних.

Вирази є комбінаціями змінних, констант та операторів, які обчислюються для отримання певного результату. Наприклад, арифметичні вирази використовуються для виконання математичних обчислень, а логічні вирази – для прийняття рішень у програмі.

Керування потоком виконання програми здійснюється за допомогою умовних операторів та циклів. Умовні оператори дозволяють виконувати різні частини коду залежно від виконання певних умов. Цикли дають змогу повторювати виконання блоків коду кілька разів, що є необхідним для обробки масивів даних або реалізації ітеративних алгоритмів.

Функції є важливим елементом структури програм у скриптових мовах. Вони дозволяють організувати код у вигляді логічно завершених блоків, які можуть багаторазово

використовуватися у різних частинах програми. Функції можуть приймати аргументи та повертати результати, що забезпечує модульність і повторне використання коду.

Структура програми у скриптових мовах зазвичай є лінійною або модульною. У простих випадках програма складається з послідовності операторів, що виконуються один за одним. У більш складних системах використовується модульний підхід, при якому програма поділяється на окремі файли або модулі, кожен з яких відповідає за певну функціональність.

Особливістю багатьох скриптових мов є чутливість до відступів або використання спеціальних символів для визначення меж блоків коду. Наприклад, у Python відступи є частиною синтаксису і визначають структуру програми, тоді як у JavaScript для цього використовуються фігурні дужки.

Коментарі є невід'ємною частиною програмного коду і використовуються для пояснення логіки роботи програми. Вони не впливають на виконання коду, але значно полегшують його розуміння та супроводження. У скриптових мовах коментарі можуть бути однорядковими або багаторядковими.

Важливим аспектом є також обробка помилок. Скриптові мови зазвичай надають механізми обробки виняткових ситуацій, які дають змогу виявляти та обробляти помилки під час виконання програми. Це дозволяє підвищити надійність програмного забезпечення та забезпечити коректну роботу навіть у випадку виникнення непередбачених ситуацій.

Інтерпретація коду є ключовим процесом у виконанні програм, написаних на скриптових мовах. Інтерпретатор послідовно аналізує та виконує кожну інструкцію програми, що забезпечує можливість інтерактивної роботи та швидкого тестування змін. Це особливо важливо у процесі розробки та налагодження програмного забезпечення.

Скриптові мови також підтримують інтеграцію з іншими технологіями. Наприклад, вони можуть використовуватися для взаємодії з базами даних, обробки файлів, роботи з мережевими протоколами або створення вебінтерфейсів. Така універсальність робить їх незамінним інструментом у сучасній програмній інженерії.

У підсумку, скриптові мови програмування є потужним засобом створення програмних систем, що поєднує простоту використання, гнучкість і високу швидкість розробки. Їх синтаксис і структура програм орієнтовані на забезпечення максимальної продуктивності праці розробника, що є критично важливим у сучасних умовах швидкого розвитку інформаційних технологій.

Лекція 2. Правила оформлення програмного коду

Оформлення програмного коду є невід'ємною складовою процесу розробки програмного забезпечення, оскільки воно безпосередньо впливає на читабельність, супроводжуваність і надійність програмних рішень. У контексті скриптових мов програмування правильна організація коду має особливе значення, адже такі мови часто використовуються для швидкої розробки, автоматизації та інтеграції систем, де зрозумілість коду визначає ефективність роботи команди.

Під правилами оформлення програмного коду розуміють сукупність узгоджених принципів і рекомендацій, що визначають структуру, стиль запису та організацію програмних елементів. Ці правила не впливають безпосередньо на виконання програми, але суттєво впливають на її якість з точки зору інженерії програмного забезпечення.

Одним із ключових принципів є узгодженість стилю. У межах одного проєкту необхідно використовувати однакові підходи до іменування змінних, функцій, класів і модулів. Наприклад, у мові Python широко застосовується стиль snake_case для імен змінних і функцій, тоді як у JavaScript частіше використовується camelCase. Узгодженість забезпечує передбачуваність структури коду та спрощує його аналіз.

Важливим аспектом є використання відступів. Відступи дозволяють візуально відобразити структуру програми, виділити вкладені блоки та підвищити зрозумілість логіки виконання. У деяких мовах, зокрема у Python, відступи мають синтаксичне значення і

визначають межі блоків коду, тоді як у інших мовах вони виконують лише роль засобу форматування.

Дотримання обмежень на довжину рядків також є важливим правилом оформлення. Занадто довгі рядки ускладнюють читання коду та його перегляд у різних середовищах розробки. Зазвичай рекомендується обмежувати довжину рядка до 80–120 символів залежно від стандартів проєкту.

Коментарі є одним із основних інструментів документування програмного коду. Вони дозволяють пояснити призначення окремих фрагментів програми, описати алгоритмічні рішення або уточнити складні місця. Важливо, щоб коментарі були змістовними, актуальними та не дублювали очевидну інформацію. Надмірна кількість коментарів, які не додають нових відомостей, може ускладнювати сприйняття коду.

Іменування ідентифікаторів має відповідати змісту та призначенню змінних і функцій. Імена повинні бути зрозумілими, уникати надмірних скорочень і відображати сутність об’єкта. Наприклад, змінна total_sum є більш інформативною, ніж змінна ts. Водночас слід уникати надто довгих імен, які ускладнюють використання коду.

Структурування коду передбачає поділ програми на логічні блоки. Це досягається за рахунок використання функцій, модулів і класів. Кожен блок повинен виконувати одну чітко визначену задачу, що відповідає принципу єдиної відповідальності. Такий підхід дає змогу підвищити повторне використання коду та спростити його тестування.

Важливим правилом є уникнення дублювання коду. Повторення однакових або подібних фрагментів коду у різних частинах програми призводить до ускладнення супроводження та підвищує ризик помилок. Для усунення дублювання використовуються функції, модулі або інші механізми абстракції.

У контексті скриптових мов особливу увагу слід приділяти обробці виняткових ситуацій. Код повинен бути організований таким чином, щоб помилки оброблялися передбачувано і не призводили до аварійного завершення програми. Для цього використовуються конструкції обробки винятків, які дозволяють локалізувати помилки та забезпечити коректне завершення роботи.

Окрім правил оформлення, важливим є розуміння лексичних елементів мови програмування, які є базовими складовими програмного коду. Лексичні елементи визначають найменші значущі одиниці мови, що використовуються для побудови синтаксичних конструкцій.

До основних лексичних елементів належать ідентифікатори. Ідентифікатор – це ім’я, що використовується для позначення змінної, функції, класу або іншого об’єкта. Ідентифікатори повинні відповідати правилам мови програмування, зокрема щодо допустимих символів і початкових символів. Зазвичай вони можуть містити літери, цифри та символ підкреслення, але не можуть починатися з цифри.

Ключові слова є зарезервованими словами мови програмування, які мають спеціальне значення і не можуть використовуватися як ідентифікатори. Наприклад, у мові Python такими словами є if, else, while, def, class. Вони визначають структуру програми та керують її виконанням.

Літерали представляють фіксовані значення, які безпосередньо використовуються у програмі. До них належать числові літерали, рядкові літерали, логічні значення та інші типи даних. Наприклад, число 10, рядок "Hello" або логічне значення True є літералами.

Оператори є спеціальними символами або словами, що виконують певні дії над даними. Вони можуть бути арифметичними, логічними, порівняння або присвоєння. Наприклад, оператор + використовується для додавання, а оператор == – для порівняння значень.

Роздільники використовуються для структуризації коду і відокремлення окремих елементів. До них належать дужки, коми, крапки з комою та інші символи. Вони допомагають визначити межі виразів, списків параметрів і блоків коду.

Коментарі також розглядаються як лексичні елементи, хоча вони не впливають на виконання програми. Інтерпретатор ігнорує коментарі, але вони є важливими для розуміння структури та логіки коду людиною.

Пробільні символи, такі як пробіли, табуляції та переходи на новий рядок, також відіграють важливу роль у лексичному аналізі. У деяких мовах вони використовуються лише для покращення читабельності, а в інших, зокрема у Python, визначають структуру програми.

Процес розбору програмного коду починається з лексичного аналізу, під час якого послідовність символів розбивається на лексеми. Лексема є конкретним екземпляром лексичного елемента. Наприклад, ім'я змінної або конкретне число є лексемою відповідного типу.

Лексичний аналіз є першим етапом обробки програмного коду і передує синтаксичному аналізу. На цьому етапі перевіряється коректність запису лексем, виявляються помилки у використанні символів і формується послідовність токенів, яка передається наступним етапам обробки.

У скриптових мовах процес лексичного аналізу часто виконується інтерпретатором безпосередньо під час виконання програми. Це забезпечує можливість швидкого виявлення помилок і інтерактивного тестування коду.

Важливою особливістю є те, що різні скриптові мови можуть мати свої специфічні правила щодо лексичних елементів. Наприклад, правила іменування, набір ключових слів або спосіб запису рядкових літералів можуть відрізнятися. Тому при роботі з конкретною мовою необхідно враховувати її особливості.

У підсумку, дотримання правил оформлення програмного коду та розуміння лексичних елементів мови програмування є основою створення якісного програмного забезпечення. Вони забезпечують зрозумілість, передбачуваність і надійність програм, що є критично важливим у професійній діяльності інженера програмного забезпечення.

Лекція 3. Змінні та їх типи

Змінні є базовим елементом будь-якої програми, оскільки вони забезпечують збереження та обробку даних під час виконання програмного коду. У скриптових мовах програмування змінна є іменованою областю пам'яті, яка містить певне значення і може змінювати його в процесі виконання програми. На відміну від багатьох компільованих мов, у скриптових мовах змінні зазвичай не потребують явного оголошення типу, що безпосередньо пов'язано з концепцією динамічної типізації.

Під типом даних розуміється множина значень і набір операцій, які можуть бути виконані над цими значеннями. Тип визначає, як саме дані зберігаються в пам'яті та які операції над ними допустимі. У скриптових мовах, таких як Python або JavaScript, існує широкий набір базових типів даних, що забезпечують гнучкість при розробці програм.

До основних типів даних належать числові типи, рядки, логічні значення та складені структури даних. Числові типи можуть бути представлені цілими числами або числами з плаваючою комою. Рядки використовуються для представлення текстової інформації, а логічні значення відображають істинність або хибність певного виразу. Складені типи, такі як списки, масиви або словники, дають змогу зберігати колекції значень.

Особливістю скриптових мов є динамічна типізація даних. Це означає, що тип змінної визначається під час виконання програми, а не на етапі її компіляції. Змінна може послідовно зберігати значення різних типів, що значно спрощує розробку та дозволяє швидко змінювати логіку програми. Наприклад, змінна може спочатку містити числове значення, а пізніше – рядок.

Динамічна типізація забезпечує високу гнучкість, але водночас потребує більшої уваги до коректності операцій. Оскільки перевірка типів виконується під час виконання, помилки, пов'язані з несумісністю типів, можуть виникати вже у процесі роботи програми. Це накладає вимоги до тестування та перевірки програмного коду.

Вирази є основним засобом обробки даних у програмі. Вираз складається зі змінних, літералів і операторів та обчислюється для отримання певного результату. Наприклад, арифметичний вираз може включати операції додавання, віднімання або множення, а логічний вираз – операції порівняння та логічні зв'язки.

Арифметичні операції використовуються для виконання математичних обчислень. До основних арифметичних операторів належать додавання, віднімання, множення, ділення та піднесення до степеня. У більшості скриптових мов також доступні операції цілочисельного ділення та знаходження залишку від ділення. Важливо враховувати, що результат виконання арифметичних операцій може залежати від типів операндів.

Логічні операції використовуються для побудови умов і прийняття рішень у програмі. Основними логічними операторами є операції логічного «і», «або» та «не». Вони застосовуються до логічних значень або виразів, результатом яких є логічні значення. Логічні операції широко використовуються у умовних конструкціях і циклах.

Операції порівняння дозволяють визначити відношення між значеннями. До них належать оператори рівності, нерівності, більший, менший та інші. Результатом таких операцій є логічне значення, яке може бути використане у подальших обчисленнях.

Рядкові операції застосовуються для обробки текстової інформації. У скриптових мовах підтримується конкатенація рядків, тобто об'єднання кількох рядків в один, а також доступ до окремих символів і підрядків. Додатково можуть використовуватися операції форматування рядків, які дозволяють вставляти значення змінних у текстові шаблони.

Особливістю роботи з рядками є те, що вони зазвичай є незмінними структурами. Це означає, що при виконанні операцій над рядками створюється новий об'єкт, а не змінюється існуючий. Такий підхід забезпечує безпечність роботи з даними, але може впливати на продуктивність при обробці великих обсягів тексту.

Перетворення типів даних є важливим аспектом роботи зі змінними. У скриптових мовах можуть використовуватися як неявні, так і явні перетворення. Неявне перетворення відбувається автоматично під час виконання операцій, коли система намагається привести значення до сумісного типу. Наприклад, числове значення може бути перетворене у рядок при виконанні операції конкатенації.

Явне перетворення виконується програмістом за допомогою спеціальних функцій або операторів. Це дає змогу точно контролювати тип даних і уникати неоднозначностей під час виконання програми. Наприклад, рядок, що містить числове значення, може бути перетворений у число для подальших обчислень.

Важливо враховувати, що не всі перетворення є коректними. Наприклад, спроба перетворити рядок, який не містить числового значення, у число призведе до помилки. Тому необхідно перевіряти коректність вхідних даних перед виконанням операцій перетворення.

У скриптових мовах також поширеним є поняття приведення типів у контексті логічних виразів. Наприклад, деякі значення можуть інтерпретуватися як істинні або хибні залежно від їх змісту. Порожній рядок або нульове значення зазвичай розглядаються як хибні, тоді як інші значення – як істинні.

Застосування операцій над даними тісно пов'язане з ефективністю програм. Правильний вибір типів даних і операцій дозволяє оптимізувати використання пам'яті та зменшити час виконання програм. У той же час неправильне використання типів може призвести до помилок або неочікуваної поведінки програми.

У сучасних скриптових мовах передбачені механізми для роботи з комплексними структурами даних, що розширює можливості обробки інформації. Наприклад, можна виконувати операції над списками, словниками або іншими колекціями, використовуючи вбудовані функції та методи.

Таким чином, змінні, типи даних, вирази та операції над даними є фундаментальними складовими програмування на скриптових мовах. Динамічна типізація забезпечує гнучкість і швидкість розробки, але потребує уважного підходу до контролю типів і перевірки коректності даних. Розуміння цих концепцій є необхідною умовою для ефективного створення програмного забезпечення.

Лекція 4. Умовні оператори

Умовні оператори є одним із базових механізмів керування потоком виконання програми, оскільки вони дають змогу реалізувати вибір альтернативних варіантів дій залежно від певних умов. У контексті скриптових мов програмування умовні конструкції використовуються для побудови розгалужених алгоритмів, що дозволяє адаптувати поведінку програми до змінних вхідних даних або стану системи.

Під умовним оператором розуміється конструкція, яка забезпечує виконання певного блоку коду лише у випадку, якщо задана логічна умова є істинною. Якщо умова не виконується, програма може або пропустити цей блок, або перейти до виконання альтернативного блоку. Таким чином, умовні оператори забезпечують нелінійний характер виконання програмного коду.

У більшості скриптових мов, зокрема Python та JavaScript, базова форма умовного оператора реалізується за допомогою конструкції if. Ця конструкція дозволяє перевірити певний логічний вираз і виконати відповідний блок коду у разі його істинності.

Логічний вираз, що використовується в умовному операторі, формується за допомогою операторів порівняння та логічних операторів. Наприклад, можна перевіряти рівність значень, їх порядок або поєднувати декілька умов за допомогою логічних зв’язків. Результатом такого виразу завжди є логічне значення, яке визначає подальший хід виконання програми.

Для реалізації альтернативних варіантів виконання використовується конструкція else. Вона визначає блок коду, який виконується у випадку, якщо основна умова не виконується. Такий підхід дозволяє забезпечити повноту обробки можливих ситуацій у програмі.

У випадках, коли необхідно перевірити декілька взаємовиключних умов, застосовується конструкція elif у Python або послідовність else if у JavaScript. Це дозволяє побудувати ланцюг перевірок, де кожна наступна умова перевіряється лише у випадку, якщо попередні не виконалися.

Реалізація розгалужених алгоритмів базується на використанні умовних операторів для вибору одного з кількох можливих шляхів виконання програми. Такий підхід широко застосовується у задачах прийняття рішень, обробки даних, валідації введених значень та керування логікою програмних систем.

Розгалужений алгоритм можна представити як структуру, у якій на певному етапі виконання відбувається перевірка умови, після чого програма переходить до одного з декількох варіантів дій. Кількість таких варіантів може бути довільною, що визначається складністю задачі.

Важливим аспектом є правильна організація умовних конструкцій. Умови повинні бути сформульовані чітко та однозначно, щоб уникнути логічних помилок. Крім того, необхідно враховувати порядок перевірки умов, оскільки він безпосередньо впливає на результат виконання програми.

Вкладені умовні конструкції використовуються у випадках, коли необхідно виконати додаткову перевірку всередині іншої умовної конструкції. Це означає, що один умовний оператор може містити всередині себе інший умовний оператор. Такий підхід дозволяє реалізувати складні логічні залежності.

Однак використання вкладених умов потребує обережності, оскільки надмірна вкладеність може ускладнювати читання та супроводження коду. У складних випадках доцільно розбивати логіку на окремі функції або використовувати інші засоби структуризації.

Прикладом вкладеної конструкції є ситуація, коли спочатку перевіряється загальна умова, а потім, у разі її істинності, виконується додаткова перевірка. Це дозволяє уточнити логіку виконання та уникнути зайвих обчислень.

Ще одним важливим інструментом є тернарний оператор, який дозволяє скорочено записати просту умовну конструкцію. Тернарний оператор є компактною формою запису умовного виразу, що повертає одне з двох значень залежно від результату перевірки умови.

У мові Python тернарний оператор має форму умовного виразу, де спочатку вказується результат для істинного випадку, потім умова, а після цього результат для хибного випадку. У JavaScript використовується класична форма condition ? value1 : value2.

Застосування тернарних операторів дозволяє зменшити обсяг коду та підвищити його компактність. Однак надмірне використання таких конструкцій, особливо у складних виразах, може негативно впливати на читабельність.

Умовні оператори також відіграють важливу роль у забезпеченні коректності програм. Вони дозволяють перевіряти вхідні дані, обробляти помилки та контролювати виконання критичних операцій. Наприклад, перед виконанням ділення доцільно перевірити, чи не дорівнює дільник нулю.

У сучасних скриптових мовах умовні конструкції можуть поєднуватися з іншими елементами мови, такими як функції, цикли та обробка винятків. Це дозволяє створювати складні алгоритми, що адаптуються до різних умов виконання.

Ефективність використання умовних операторів значною мірою залежить від якості побудови логічних виразів. Необхідно уникати надмірно складних умов, які важко аналізувати. У таких випадках доцільно розбивати умови на простіші складові або використовувати допоміжні змінні.

З точки зору оптимізації, важливо враховувати, що перевірка умов також потребує обчислювальних ресурсів. Тому у критичних ділянках коду слід організовувати перевірки таким чином, щоб мінімізувати кількість зайвих обчислень.

У підсумку, умовні оператори є фундаментальним засобом реалізації логіки програм у скриптових мовах. Вони забезпечують можливість створення розгалужених алгоритмів, дозволяють реалізувати складні залежності та адаптувати поведінку програмного забезпечення до різних умов. Правильне використання умовних конструкцій є важливою складовою професійної діяльності інженера програмного забезпечення.

Лекція 5. Цикли у скриптових мовах, їх види, організація ітерацій, оператори керування виконанням та побудова алгоритмів із використанням базових конструкцій

Цикли є одним із ключових механізмів керування потоком виконання програми, оскільки вони дають змогу багаторазово виконувати певний блок коду. У скриптових мовах програмування цикли широко застосовуються для обробки даних, реалізації алгоритмів, автоматизації обчислень та організації повторюваних дій. Використання циклів дозволяє уникнути дублювання коду і забезпечити компактність програмних рішень.

Під циклом розуміється конструкція, що забезпечує повторення виконання блоку команд за умови виконання певного логічного виразу або до досягнення заданого стану. Кожне повторення виконання блоку називається ітерацією. Кількість ітерацій може бути визначеною або невизначеною, що залежить від умов задачі.

У скриптових мовах, зокрема Python та JavaScript, існує кілька основних видів циклів, які відрізняються способом організації перевірки умови та керування ітераціями.

Цикл з передумовою передбачає перевірку умови перед виконанням тіла циклу. Якщо умова є істинною, тіло циклу виконується, після чого перевірка повторюється. Якщо умова з самого початку є хибною, тіло циклу не виконується жодного разу. Такий тип циклу реалізується за допомогою конструкції while. Він використовується у випадках, коли кількість ітерацій заздалегідь невідома і визначається під час виконання програми.

Особливістю циклів з передумовою є необхідність забезпечення коректного завершення. Якщо умова циклу завжди залишається істинною, виникає нескінченний цикл, який може призвести до зависання програми. Тому важливо передбачити зміну стану, що впливає на умову завершення.

Цикл з післяумовою передбачає перевірку умови після виконання тіла циклу. Це означає, що тіло циклу виконується принаймні один раз незалежно від початкового значення умови. У деяких мовах програмування такий цикл реалізується конструкцією do-while, яка є характерною для JavaScript. У Python прямої конструкції циклу з післяумовою немає, але аналогічна поведінка може бути реалізована іншими засобами.

Ще одним поширеним видом є цикл з лічильником, який використовується для виконання заданої кількості ітерацій. У скриптових мовах він часто реалізується у вигляді конструкції for. Такий цикл дозволяє зручно організувати перебір послідовностей або діапазонів значень.

Ітерація по колекціях є важливою особливістю сучасних скриптових мов. Колекції, такі як списки, масиви, множини або словники, можуть бути оброблені за допомогою циклу for, який послідовно перебирає їх елементи. Це значно спрощує роботу з наборами даних і дозволяє реалізувати обробку без явного використання індексів.

У Python ітерація по колекціях реалізується через механізм ітераторів, що дозволяє абстрагуватися від внутрішньої структури даних. У JavaScript також існують різні варіанти ітерації, включаючи класичний цикл for та спеціалізовані конструкції для роботи з об'єктами та масивами.

Оператори керування виконанням циклів дозволяють змінювати стандартний хід виконання ітерацій. До таких операторів належать break, continue та інші подібні конструкції. Оператор break забезпечує негайне завершення виконання циклу незалежно від умови. Це може бути корисно, коли досягнуто необхідний результат або виникла помилка.

Оператор continue дозволяє пропустити поточну ітерацію і перейти до наступної перевірки умови. Це дає змогу уникнути виконання частини коду для певних значень і спростити логіку програми. Використання таких операторів повинно бути обґрунтованим, оскільки надмірне їх застосування може ускладнювати розуміння коду.

Організація керування виконанням програм базується на поєднанні базових конструкцій: послідовності, розгалуження та повторення. Послідовність передбачає виконання інструкцій одна за одною, розгалуження – вибір одного з кількох варіантів, а повторення – багаторазове виконання певного блоку коду.

Побудова алгоритмів із використанням базових конструкцій дозволяє формалізувати процес розв'язання задачі. Будь-який алгоритм може бути представлений у вигляді комбінації цих трьох конструкцій. Це забезпечує універсальність підходу до розробки програм і спрощує їх аналіз.

При розробці алгоритмів важливо враховувати ефективність використання циклів. Надмірна кількість ітерацій або неефективна організація циклів може призвести до значного збільшення часу виконання програми. Тому необхідно оптимізувати умови завершення та мінімізувати обсяг обчислень у тілі циклу.

Особливу увагу слід приділяти вкладеним циклам, коли один цикл знаходиться всередині іншого. У таких випадках кількість виконань внутрішнього циклу залежить від кількості ітерацій зовнішнього, що може призвести до квадратичного або навіть більшого зростання складності алгоритму. Це є критично важливим при роботі з великими обсягами даних.

У скриптових мовах також широко використовуються конструкції генерації послідовностей, які дозволяють компактно створювати колекції даних. Наприклад, у Python існують генератори списків, що дозволяють поєднувати ітерацію та обчислення в одному виразі.

Контроль правильності роботи циклів є важливим етапом розробки програм. Необхідно перевіряти коректність умов завершення, уникати нескінченних циклів і забезпечувати правильну обробку граничних випадків. Для цього використовуються тестування та аналіз логіки виконання.

Таким чином, цикли є фундаментальним інструментом реалізації алгоритмів у скриптових мовах програмування. Вони забезпечують можливість організації повторюваних дій, обробки колекцій даних і побудови складних алгоритмічних структур. Правильне використання циклів та операторів керування виконанням дозволяє створювати ефективні, зрозумілі та надійні програмні рішення.

Лекція 6. Функції у скриптових мовах

Функції є одним із ключових інструментів структуризації програмного коду, оскільки вони дозволяють виділяти логічно завершені фрагменти програми в окремі модулі, які можуть багаторазово використовуватися. У скриптових мовах програмування функції відіграють особливо важливу роль, адже забезпечують компактність коду, підвищують його читабельність і спрощують супроводження програмних систем.

Під функцією розуміється іменований блок програмного коду, який виконує певну задачу і може бути викликаний з інших частин програми. Функції дозволяють реалізувати принцип модульності, згідно з яким складна система розбивається на простіші складові. Це дає змогу зменшити складність розробки та підвищити якість програмного забезпечення.

Основним призначенням функцій є повторне використання коду. Якщо певна логіка використовується у кількох місцях програми, доцільно винести її у функцію. Це дозволяє уникнути дублювання коду та спрощує внесення змін. У разі необхідності змінити алгоритм достатньо внести зміни лише у функції, а не у всіх місцях її використання.

Оголошення функції передбачає визначення її імені, списку параметрів і тіла функції. У скриптових мовах, таких як Python, для оголошення функції використовується спеціальна конструкція, яка визначає початок функції та її структуру. У JavaScript існує кілька способів оголошення функцій, включаючи класичні функції та функціональні вирази.

Виклик функції здійснюється шляхом звернення до її імені з передачею необхідних аргументів. Під час виклику функції відбувається передача керування у її тіло, виконання інструкцій і, за необхідності, повернення результату. Після завершення виконання функції програма продовжує роботу з того місця, де було здійснено виклик.

Передача параметрів у функції є важливим аспектом їх використання. Параметри дозволяють передавати у функцію дані, які використовуються під час її виконання. Аргументи, що передаються у функцію під час виклику, співставляються з параметрами, визначеними при її оголошенні.

Існують різні способи передачі параметрів, серед яких найбільш поширеними є передача за значенням і передача за посиланням. При передачі за значенням у функцію передається копія значення змінної, і зміни, виконані у функції, не впливають на початкову змінну. При передачі за посиланням у функцію передається доступ до самого об’єкта, і зміни можуть впливати на вихідні дані.

У скриптових мовах реалізація передачі параметрів може мати особливості. Наприклад, у Python використовується модель передачі посилання на об’єкт, що поєднує риси обох підходів. У JavaScript примітивні типи передаються за значенням, а об’єкти – за посиланням.

Іменовані та позиційні параметри визначають спосіб передачі аргументів у функцію. Позиційні параметри передаються у тому порядку, у якому вони визначені у сигнатурі функції. Іменовані параметри дозволяють явно вказувати ім’я параметра при передачі значення, що підвищує читабельність коду та зменшує ймовірність помилок.

Використання іменованих параметрів є особливо доцільним у функціях з великою кількістю параметрів або параметрами з однаковими типами. Це дозволяє уникнути плутанини і зробити код більш зрозумілим.

Значення параметрів за замовчуванням дозволяють визначити стандартні значення для параметрів функції. Якщо під час виклику функції відповідний аргумент не передано, використовується значення за замовчуванням. Це спрощує виклик функцій і дозволяє зменшити кількість обов’язкових параметрів.

Однак при використанні значень за замовчуванням необхідно враховувати особливості мови програмування. У деяких випадках значення за замовчуванням можуть обчислюватися лише один раз під час оголошення функції, що може призводити до неочікуваної поведінки при роботі з змінюваними об’єктами.

Повернення результатів виконання функції здійснюється за допомогою спеціальної конструкції, яка передає значення з функції у місце її виклику. Функція може повертати як одне значення, так і кілька значень, залежно від можливостей мови програмування.

Якщо функція не містить явного повернення результату, вона може повертати спеціальне значення, яке означає відсутність результату. Це необхідно враховувати при використанні функцій у виразах.

Функції можуть використовуватися не лише для обчислення значень, але й для виконання дій, таких як обробка даних, взаємодія з користувачем або робота з файлами. У таких випадках результат виконання може бути неявним і проявлятися у зміні стану системи.

Важливим аспектом є область видимості змінних, що використовуються у функціях. Локальні змінні визначаються всередині функції і доступні лише у її межах, тоді як глобальні змінні можуть бути доступні з різних частин програми. Правильне використання областей видимості дозволяє уникнути конфліктів і підвищити надійність програм.

Функції також можуть бути вкладеними або передаватися як аргументи іншим функціям, що є характерною особливістю багатьох скриптових мов. Це відкриває можливості для реалізації функціонального підходу до програмування.

Застосування функцій є основою побудови складних програмних систем. Вони дозволяють організувати код у вигляді зрозумілих і керованих модулів, що спрощує розробку, тестування та супроводження програмного забезпечення.

Таким чином, функції є універсальним засобом реалізації алгоритмів у скриптових мовах програмування. Вони забезпечують модульність, повторне використання коду і гнучкість у побудові програм. Розуміння принципів оголошення, виклику та передачі параметрів у функції є необхідною умовою ефективної роботи програміста.

Лекція 7. Області видимості змінних

Області видимості змінних є фундаментальним поняттям програмування, яке визначає, у яких частинах програми доступна певна змінна та де вона може бути використана. У скриптових мовах програмування правильне розуміння областей видимості є критично важливим для побудови коректних і передбачуваних програм, оскільки воно безпосередньо впливає на поведінку змінних під час виконання.

Під областю видимості змінної розуміється частина програмного коду, у межах якої змінна є доступною. Області видимості визначаються структурою програми, зокрема функціями, блоками коду та модулями. Основною метою використання областей видимості є ізоляція змінних і запобігання конфліктам імен.

У більшості скриптових мов, таких як Python та JavaScript, виділяють локальні та глобальні області видимості. Ці два типи областей визначають рівень доступності змінних у програмі.

Локальні змінні визначаються всередині функції або блоку коду і доступні лише в межах цього блоку. Вони створюються під час виклику функції і знищуються після завершення її виконання. Використання локальних змінних дозволяє уникнути небажаного впливу на інші частини програми і забезпечує ізоляцію логіки.

Глобальні змінні визначаються поза функціями і доступні у всіх частинах програми. Вони існують протягом усього часу виконання програми. Використання глобальних змінних може спрощувати доступ до спільних даних, але водночас підвищує ризик виникнення помилок, пов’язаних із неконтрольованими змінами їх значень.

У Python для зміни глобальної змінної всередині функції необхідно явно вказати її як глобальну. У JavaScript область видимості також залежить від способу оголошення змінної, що впливає на її доступність.

Окрім локальної та глобальної областей, у деяких випадках виділяють вкладені області видимості. Це виникає, коли функції визначаються всередині інших функцій. У таких ситуаціях внутрішня функція може мати доступ до змінних зовнішньої функції, що забезпечує додаткову гнучкість у побудові програм.

Правильне використання областей видимості дозволяє уникнути конфліктів імен, зменшити складність програм і підвищити їх надійність. Рекомендується мінімізувати використання глобальних змінних і надавати перевагу локальним змінним, що обмежують область впливу змін.

Рекурсія є ще одним важливим механізмом у програмуванні, який дозволяє функції викликати саму себе. Такий підхід використовується для розв'язання задач, які можуть бути представлені у вигляді підзадач того самого типу. Рекурсія часто застосовується у задачах обробки деревоподібних структур, пошуку, сортування та інших алгоритмах.

Під рекурсивною функцією розуміється функція, яка у процесі свого виконання викликає саму себе з новими параметрами. Кожен такий виклик створює новий контекст виконання, що дозволяє розв'язувати задачу поетапно.

Ключовими елементами рекурсії є базовий випадок і рекурсивний випадок. Базовий випадок визначає умову завершення рекурсії і запобігає нескінченному виклику функції. Рекурсивний випадок описує, як задача розбивається на підзадачі і як виконується подальший виклик функції.

Без наявності базового випадку рекурсія призведе до нескінченного виклику функції, що, у свою чергу, викличе переповнення стеку викликів і аварійне завершення програми. Тому правильне формулювання базового випадку є критично важливим.

Рекурсія дозволяє компактно описати алгоритми, які мають природну ієрархічну структуру. Наприклад, обхід дерев, обчислення факторіала або реалізація алгоритмів розділяй і володарюй можуть бути ефективно реалізовані за допомогою рекурсії.

Разом із тим рекурсія має певні недоліки, пов'язані з використанням пам'яті та продуктивністю. Кожен виклик функції створює новий запис у стеку, що може призвести до значних витрат ресурсів при великій глибині рекурсії. У таких випадках доцільно розглядати альтернативні підходи.

Ітерація є альтернативою рекурсії і передбачає використання циклів для повторення обчислень. Ітеративні алгоритми зазвичай є більш ефективними з точки зору використання пам'яті, оскільки не потребують створення великої кількості контекстів виконання.

Порівняння рекурсії та ітерації дозволяє визначити доцільність використання кожного підходу. Рекурсія є більш природною для задач, що мають рекурсивну структуру, і дозволяє отримати компактний і зрозумілий код. Ітерація, у свою чергу, забезпечує кращу продуктивність і контроль над виконанням.

У скриптових мовах, таких як Python, рекурсія підтримується на рівні мови, але має обмеження на глибину викликів. У JavaScript також існують обмеження, пов'язані з розміром стеку викликів.

При виборі між рекурсією та ітерацією необхідно враховувати складність задачі, обсяг даних і вимоги до продуктивності. У деяких випадках можливе поєднання обох підходів для досягнення оптимального результату.

Таким чином, області видимості змінних і рекурсія є важливими концепціями, що визначають структуру та поведінку програм у скриптових мовах. Розуміння цих механізмів дозволяє ефективно організовувати програмний код, уникати помилок і обирати оптимальні підходи до розв'язання задач.

Лекція 8. Модульна організація програм

Модульна організація програм є одним із базових принципів сучасної програмної інженерії, оскільки вона забезпечує структурованість, повторне використання коду та зручність супроводження програмних систем. У скриптових мовах програмування модулі дозволяють розбивати програму на окремі логічні частини, кожна з яких реалізує певну функціональність.

Під модулем розуміється окремий файл або логічна одиниця програмного коду, що містить змінні, функції, класи та інші елементи, які можуть бути використані в інших

частинах програми. Модулі дозволяють організувати код таким чином, щоб кожна частина відповідала за окрему задачу, що відповідає принципу єдиної відповідальності.

Основною перевагою модульного підходу є можливість повторного використання коду. Один і той самий модуль може бути підключений у різних проєктах або частинах програми, що значно скорочує час розробки та підвищує якість програмного забезпечення. Крім того, модулі дозволяють розподіляти роботу між кількома розробниками.

Імпорт модулів є механізмом підключення зовнішнього коду до поточної програми. У скриптових мовах, таких як Python та JavaScript, імпорт дозволяє використовувати функції, класи та змінні, визначені в інших файлах.

У Python імпорт здійснюється за допомогою спеціальних конструкцій, які дозволяють підключати модуль повністю або лише окремі його елементи. У JavaScript також існують сучасні механізми імпорту, що дозволяють організувати модульну структуру програм.

Імпорт може виконуватися різними способами. Повний імпорт дозволяє отримати доступ до всіх елементів модуля через його ім’я. Частковий імпорт дозволяє підключити лише окремі елементи, що зменшує обсяг доступного простору імен і підвищує зрозумілість коду.

Важливим аспектом є використання псевдонімів при імпорті. Це дозволяє скорочувати імена модулів або уникати конфліктів імен. Наприклад, модуль може бути імпортований під коротшим ім’ям, що спрощує його використання у програмі.

Простори імен є механізмом організації ідентифікаторів у програмі. Вони визначають, які імена доступні у певному контексті і як вони співвідносяться між собою. Простір імен можна розглядати як словник, який відображає імена на відповідні об’єкти.

У скриптових мовах існує кілька рівнів просторів імен, включаючи локальний, глобальний та простір імен модуля. Кожен з цих рівнів має свої правила доступу і визначає, які змінні та функції можуть бути використані у конкретному місці програми.

Простори імен дозволяють уникнути конфліктів імен, коли різні частини програми використовують однакові ідентифікатори. Завдяки модульній організації кожен модуль має власний простір імен, що ізолює його внутрішні елементи від інших частин програми.

Пошук імен у програмі здійснюється за певними правилами. Спочатку перевіряється локальний простір імен, потім – зовнішні області, а далі – глобальний простір імен. Такий підхід забезпечує передбачуваність доступу до змінних і функцій.

Створення власних модулів є важливим етапом організації програмного коду. Для цього достатньо створити окремий файл і визначити у ньому необхідні функції, класи або змінні. Після цього модуль може бути імпортований у інші частини програми.

При створенні модулів важливо дотримуватися принципів структурованості та зрозумілості. Кожен модуль повинен виконувати чітко визначену задачу і мати логічно завершений зміст. Імена модулів повинні відображати їх призначення.

У великих програмних системах модулі можуть об’єднуватися у пакети, що дозволяє створювати ієрархічну структуру проєкту. Це спрощує навігацію по коду і дозволяє організувати великі обсяги програмного коду у зрозумілу структуру.

Важливим аспектом є контроль доступу до елементів модуля. У деяких мовах існують механізми, що дозволяють визначити, які елементи модуля є доступними для зовнішнього використання, а які залишаються внутрішніми.

Модульна організація також сприяє тестуванню програм. Кожен модуль може бути протестований окремо, що дозволяє виявляти помилки на ранніх етапах розробки. Це підвищує надійність програмного забезпечення.

У скриптових мовах модулі часто використовуються для підключення бібліотек і фреймворків, що значно розширює можливості програмування. Це дозволяє використовувати готові рішення для типових задач і зосередитися на реалізації специфічної логіки.

Таким чином, модульна організація програм є важливим інструментом підвищення якості програмного забезпечення. Вона забезпечує структурованість, повторне використання коду та зручність супроводження. Розуміння принципів імпорту модулів, роботи з

просторами імен і створення власних модулів є необхідною складовою підготовки інженера програмного забезпечення.

Лекція 9. Організація пакетів у скриптових мовах

Організація програмного коду у вигляді пакетів є логічним розвитком модульного підходу і застосовується у випадках, коли обсяг програмної системи зростає настільки, що використання окремих модулів стає недостатнім для забезпечення структурованості. Пакети дозволяють об'єднувати пов'язані модулі у єдину ієрархічну структуру, що значно спрощує навігацію, супроводження та масштабування програмних систем.

Під пакетом розуміється сукупність модулів, організованих у вигляді каталогу або простору імен, що відображає логічну структуру програмного забезпечення. Кожен пакет може містити підпакети, що дозволяє створювати багаторівневу ієрархію. Такий підхід забезпечує впорядкування коду відповідно до функціонального призначення його складових.

У скриптових мовах, таких як Python та JavaScript, пакети реалізуються як директорії, що містять модулі та службові файли. У Python пакет зазвичай ідентифікується наявністю спеціального файлу, який визначає його як окрему одиницю. У JavaScript організація пакетів пов'язана з використанням систем керування залежностями.

Ієрархічна структура пакетів дозволяє логічно групувати функціональні компоненти. Наприклад, окремі пакети можуть відповідати за обробку даних, взаємодію з мережею або реалізацію інтерфейсу користувача. Такий підхід забезпечує розподіл відповідальності та підвищує зрозумілість структури програми.

Повторне використання програмного коду є одним із ключових принципів програмної інженерії. Воно передбачає застосування вже реалізованих компонентів у нових задачах без необхідності їх повторного створення. Пакети і модулі виступають основним механізмом забезпечення повторного використання коду.

Повторне використання дозволяє значно скоротити час розробки, зменшити кількість помилок і підвищити надійність програмного забезпечення. Використання перевірених компонентів знижує ризики, пов'язані з реалізацією нових функцій, і дозволяє зосередитися на вирішенні специфічних задач.

Існують різні рівні повторного використання коду. Найпростішим є використання функцій або модулів у межах одного проєкту. Більш складним є повторне використання пакетів у різних проєктах або навіть у різних програмних системах. Це вимагає дотримання певних стандартів і правил організації коду.

Для ефективного повторного використання необхідно забезпечити незалежність модулів і пакетів. Компоненти повинні мати чітко визначені інтерфейси і мінімальні залежності від інших частин системи. Це дозволяє використовувати їх у різних контекстах без необхідності значних змін.

Принципи модульності визначають основні підходи до організації програмного коду. Одним із ключових принципів є принцип єдиної відповідальності, згідно з яким кожен модуль або пакет повинен виконувати одну конкретну функцію. Це спрощує розуміння і тестування коду.

Іншим важливим принципом є низька зв'язаність і висока згуртованість. Низька зв'язаність означає, що модулі повинні бути максимально незалежними один від одного, а висока згуртованість – що всі елементи модуля повинні бути тісно пов'язані між собою і виконувати спільну задачу.

Інкапсуляція є ще одним важливим принципом модульності. Вона передбачає приховування внутрішньої реалізації модуля і надання доступу лише до необхідних елементів через чітко визначений інтерфейс. Це дозволяє змінювати внутрішню реалізацію без впливу на інші частини програми.

Абстракція дозволяє виділити суттєві характеристики системи і приховати деталі реалізації. У контексті модульності це означає, що користувач модуля взаємодіє лише з його інтерфейсом, не заглиблюючись у внутрішню структуру.

Ієрархічна організація пакетів також сприяє реалізації принципів модульності. Вона дозволяє будувати складні системи з простіших компонентів, що взаємодіють між собою через визначені інтерфейси. Це забезпечує масштабованість і гнучкість програмного забезпечення.

Важливим аспектом є управління залежностями між модулями і пакетами. Надмірна кількість залежностей може ускладнювати супроводження і призводити до проблем при зміні окремих компонентів. Тому необхідно прагнути до мінімізації залежностей і чіткої їх організації.

У сучасних скриптових мовах підтримуються механізми підключення зовнішніх пакетів, що дозволяє використовувати готові бібліотеки для розв'язання типових задач. Це значно розширює можливості програмування і підвищує продуктивність розробки.

Організація пакетів також відіграє важливу роль у тестуванні програм. Чітке розділення коду на модулі і пакети дозволяє проводити модульне тестування, що забезпечує виявлення помилок на ранніх етапах розробки.

Таким чином, організація програм у вигляді пакетів, повторне використання коду та дотримання принципів модульності є основою створення якісного програмного забезпечення. Вони забезпечують структурованість, гнучкість і масштабованість програмних систем, що є необхідним для ефективної роботи інженера програмного забезпечення.

Лекція 10. Використання стандартних і сторонніх бібліотек

У сучасній програмній інженерії ефективна розробка програмного забезпечення неможлива без використання бібліотек, які надають готові реалізації типових задач. Бібліотеки дозволяють значно скоротити час розробки, підвищити якість програмного коду та уникнути повторної реалізації вже відомих алгоритмів і механізмів. У скриптових мовах програмування бібліотеки відіграють особливо важливу роль завдяки їх гнучкості та широким можливостям інтеграції.

Під бібліотекою розуміється набір функцій, класів і модулів, які можуть бути використані у програмі для виконання певних задач. Бібліотеки можуть бути стандартними, тобто такими, що входять до складу мови програмування, або сторонніми, створеними незалежними розробниками.

Стандартні бібліотеки є невід'ємною частиною середовища програмування і постачаються разом із мовою. Вони містять базові засоби для роботи з файлами, рядками, математичними обчисленнями, мережевими протоколами та іншими аспектами програмування. Наприклад, у Python стандартна бібліотека є досить об'ємною і дозволяє реалізувати широкий спектр задач без залучення зовнішніх ресурсів.

Використання стандартних бібліотек здійснюється шляхом імпорту відповідних модулів у програму. Це дозволяє отримати доступ до функцій і класів, що реалізують необхідну функціональність. Перевагою стандартних бібліотек є їх стабільність, перевіреність і сумісність із мовою програмування.

Однак стандартних бібліотек не завжди достатньо для реалізації складних або специфічних задач. У таких випадках використовуються сторонні бібліотеки, які розробляються спільнотою або окремими організаціями. Вони можуть надавати розширені можливості, такі як робота з базами даних, машинне навчання, обробка зображень або створення вебзастосунків.

Підключення сторонніх бібліотек передбачає їх встановлення у середовище розробки та подальший імпорт у програму. У Python для цього використовуються спеціальні інструменти керування пакетами, що дозволяють завантажувати і встановлювати бібліотеки

з централізованих сховищ. У JavaScript аналогічні механізми реалізуються через системи керування пакетами.

Сторонні бібліотеки дозволяють значно розширити функціональність програмного забезпечення, але їх використання потребує обережності. Необхідно враховувати надійність джерела бібліотеки, її сумісність з версією мови програмування та можливі залежності від інших компонентів.

Керування залежностями є важливим аспектом організації програмного проєкту. Під залежностями розуміються зовнішні бібліотеки та модулі, від яких залежить робота програми. З часом кількість залежностей може зростати, що ускладнює їх контроль і підтримку.

Основною задачею керування залежностями є забезпечення узгодженості версій бібліотек і їх коректної взаємодії. Використання різних версій однієї і тієї ж бібліотеки може призвести до конфліктів і помилок під час виконання програми. Тому необхідно фіксувати версії залежностей і контролювати їх оновлення.

У сучасних системах розробки використовуються спеціальні інструменти для керування залежностями. Вони дозволяють автоматизувати процес встановлення, оновлення і видалення бібліотек, а також забезпечують відтворюваність середовища виконання програми. Це є особливо важливим при командній розробці, коли необхідно забезпечити однакові умови для всіх учасників проєкту.

Ізоляція середовища виконання є ще одним важливим аспектом керування залежностями. Вона дозволяє створювати окремі середовища для різних проєктів, що запобігає конфліктам між бібліотеками. У Python для цього використовуються віртуальні середовища, а у JavaScript – локальні каталоги залежностей.

Файл конфігурації залежностей є важливим елементом проєкту. Він містить перелік бібліотек і їх версій, необхідних для роботи програми. Це дозволяє швидко відновити середовище виконання і забезпечити коректну роботу програми на різних системах.

Оновлення залежностей повинно виконуватися з урахуванням можливих змін у функціональності бібліотек. Нові версії можуть містити як покращення, так і зміни, що порушують сумісність. Тому перед оновленням необхідно проводити тестування і перевірку роботи програми.

Важливим аспектом є також безпека використання бібліотек. Сторонні компоненти можуть містити вразливості або небезпечний код. Тому необхідно використовувати перевірені джерела і регулярно перевіряти залежності на наявність відомих проблем.

Документування залежностей є необхідною умовою для підтримки програмного проєкту. Усі використовувані бібліотеки повинні бути чітко зазначені, щоб забезпечити прозорість і можливість відтворення середовища виконання.

Таким чином, використання стандартних і сторонніх бібліотек є важливою складовою сучасного програмування. Вони дозволяють значно розширити можливості програм і підвищити ефективність розробки. Водночас керування залежностями є критично важливим для забезпечення стабільності, безпеки та відтворюваності програмного забезпечення.

Лекція 11. Класи та об’єкти

Об’єктно-орієнтований підхід є одним із базових підходів до розробки програмного забезпечення, який дозволяє моделювати реальні системи у вигляді взаємодіючих об’єктів. У скриптових мовах програмування цей підхід широко використовується завдяки гнучкості

мови та підтримці динамічної типізації. Ключовими поняттями об’єктно-орієнтованого програмування є клас і об’єкт.

Клас є шаблоном або описом структури та поведінки об’єктів. Він визначає, які дані може містити об’єкт і які операції над ними можуть виконуватися. Об’єкт, у свою чергу, є конкретним екземпляром класу, що має власний стан і може виконувати визначені дії.

У скриптових мовах, таких як Python та JavaScript, класи використовуються для організації коду і створення складних програмних структур. Об’єкти дозволяють об’єднати дані і функціональність в єдину сутність, що спрощує моделювання предметної області.

Атрибути класу визначають дані, які зберігаються в об’єкті. Вони можуть бути різних типів і представляють стан об’єкта. Атрибути можуть бути як загальними для всіх об’єктів класу, так і унікальними для кожного окремого екземпляра.

Методи класу визначають поведінку об’єктів. Вони є функціями, що належать класу і можуть працювати з його атрибутами. Методи дозволяють реалізувати логіку обробки даних і взаємодії між об’єктами.

Ініціалізація об’єктів є процесом створення нового екземпляра класу і встановлення його початкового стану. Під час ініціалізації визначаються значення атрибутів, що характеризують об’єкт. Це дозволяє забезпечити коректність роботи об’єкта з моменту його створення.

Конструктор є спеціальним методом, який виконується під час створення об’єкта. Його призначення полягає у ініціалізації атрибутів і підготовці об’єкта до використання. У Python конструктор реалізується через спеціальний метод, який автоматично викликається при створенні об’єкта. У JavaScript також існують механізми для визначення конструктора.

Інкапсуляція є одним із основних принципів об’єктно-орієнтованого програмування. Вона передбачає об’єднання даних і методів, що працюють з цими даними, в межах одного класу, а також обмеження прямого доступу до внутрішнього стану об’єкта. Це дозволяє захистити дані від некоректного використання і забезпечити контроль над їх зміною.

Приховування даних є складовою інкапсуляції і передбачає обмеження доступу до атрибутів класу. Деякі атрибути можуть бути доступні лише всередині класу, тоді як інші можуть бути відкриті для зовнішнього використання. Це дозволяє забезпечити цілісність даних і уникнути неконтрольованих змін.

У скриптових мовах реалізація приховування даних може мати різні форми. Наприклад, у Python використовуються умовні домовленості щодо іменування атрибутів, які сигналізують про їх обмежений доступ. У JavaScript існують механізми, що дозволяють визначати приватні властивості.

Методи доступу до атрибутів використовуються для отримання і зміни значень атрибутів об’єкта. Вони дозволяють контролювати доступ до даних і забезпечити перевірку коректності значень. Такі методи часто називають гетерами і сетерами.

Використання методів доступу дозволяє змінювати внутрішню реалізацію класу без впливу на зовнішній код. Це забезпечує гнучкість і спрощує супроводження програмного забезпечення. Наприклад, замість прямого доступу до атрибута можна використовувати метод, який виконує додаткову перевірку.

Важливим аспектом є зв’язок між об’єктами. Об’єкти можуть взаємодіяти між собою шляхом виклику методів і передачі даних. Це дозволяє будувати складні системи з простіших компонентів.

Об’єктно-орієнтований підхід сприяє повторному використанню коду. Класи можуть бути використані для створення великої кількості об’єктів, що мають однакову структуру і поведінку. Це зменшує обсяг коду і підвищує ефективність розробки.

При розробці класів важливо дотримуватися принципів проєктування, таких як зрозумілість, модульність і мінімізація залежностей. Кожен клас повинен мати чітко визначене призначення і не містити зайвої функціональності.

Таким чином, класи і об’єкти є основою об’єктно-орієнтованого програмування у скриптових мовах. Вони дозволяють організувати програмний код у вигляді взаємодіючих компонентів, що мають власний стан і поведінку. Інкапсуляція та методи доступу до атрибутів забезпечують контроль над даними і підвищують надійність програмного забезпечення.

Лекція 12. Організація програмних компонентів

Організація програмних компонентів є ключовим етапом розробки програмного забезпечення, оскільки саме від структури компонентів залежить зрозумілість, масштабованість і супроводжуваність системи. У скриптових мовах програмування компоненти формуються на основі класів, функцій і модулів, що дозволяє будувати складні системи з відносно простих елементів.

Під програмним компонентом розуміється логічно завершена частина програмного забезпечення, яка виконує певну функцію і може взаємодіяти з іншими компонентами через визначені інтерфейси. Компоненти можуть бути різного рівня складності – від окремих функцій до цілих підсистем.

Класи відіграють центральну роль в організації компонентів, оскільки вони дозволяють об’єднати дані та поведінку в єдину структуру. Використання класів дає змогу реалізувати принципи модульності, інкапсуляції та повторного використання коду, що є основою сучасного програмування.

Створення класу передбачає визначення його структури, включаючи атрибути та методи. Атрибути визначають стан об’єкта, а методи – операції, які можуть бути виконані над цими даними. У скриптових мовах, таких як Python та JavaScript, класи створюються за допомогою спеціальних конструкцій, що визначають їх поведінку.

При проєктуванні класів важливо враховувати їх призначення та відповідальність. Кожен клас повинен виконувати одну чітко визначену функцію, що відповідає принципу єдиної відповідальності. Це дозволяє зменшити складність системи і спростити її супроводження.

Використання класів передбачає створення їх екземплярів, тобто об’єктів. Екземпляр класу є конкретною реалізацією класу, що має власний стан і може виконувати визначені методи. Процес створення екземпляра називається інстанціюванням.

Під час створення об’єкта викликається конструктор класу, який ініціалізує його атрибути. Це дозволяє задати початковий стан об’єкта і підготувати його до використання. Значення атрибутів можуть передаватися у конструктор як параметри.

Робота з екземплярами класів передбачає доступ до їх атрибутів і виклик методів. Атрибути можуть зчитуватися або змінюватися, а методи – виконувати певні дії. Це дозволяє реалізувати поведінку об’єкта відповідно до логіки програми.

У скриптових мовах доступ до атрибутів і методів здійснюється через спеціальний синтаксис, який визначає зв’язок між об’єктом і його властивостями. Це забезпечує зручність використання об’єктів і зрозумілість коду.

Взаємодія між об’єктами є важливим аспектом організації програмних компонентів. Об’єкти можуть обмінюватися даними, викликати методи один одного і спільно виконувати складні задачі. Такий підхід дозволяє будувати гнучкі та масштабовані системи.

Компонентний підхід передбачає, що система складається з незалежних, але взаємодіючих частин. Це дозволяє змінювати або замінювати окремі компоненти без впливу на всю систему. У результаті підвищується гнучкість і зменшується складність розробки.

Важливим аспектом є повторне використання класів і компонентів. Один і той самий клас може бути використаний для створення багатьох об’єктів, що мають однакову структуру, але різні значення атрибутів. Це дозволяє ефективно використовувати ресурси і зменшити обсяг коду.

Організація компонентів також пов’язана з використанням модулів і пакетів. Класи можуть бути розміщені у різних модулях, що дозволяє структурувати програму і розділити її на логічні частини. Це особливо важливо для великих проєктів.

При розробці програмних компонентів необхідно враховувати принципи взаємодії, такі як слабка зв’язаність і чітко визначені інтерфейси. Компоненти повинні взаємодіяти між собою через обмежений набір методів, що зменшує залежності і підвищує надійність системи.

Тестування компонентів є важливою частиною процесу розробки. Кожен клас і його методи повинні бути перевірені окремо, щоб забезпечити їх коректну роботу. Це дозволяє виявляти помилки на ранніх етапах і підвищує якість програмного забезпечення.

У сучасних скриптових мовах також підтримується динамічне створення об’єктів і зміна їх властивостей під час виконання програми. Це забезпечує додаткову гнучкість і дозволяє адаптувати поведінку системи до змінних умов.

Таким чином, організація програмних компонентів на основі класів і об’єктів є ефективним підходом до розробки програмного забезпечення. Вона забезпечує структурованість, повторне використання коду і гнучкість системи. Розуміння принципів створення класів і роботи з їх екземплярами є необхідною умовою для ефективної діяльності інженера програмного забезпечення.

Лекція 13. Наслідування, перевизначення методів

Наслідування є одним із базових механізмів об’єктно-орієнтованого програмування, який дозволяє створювати нові класи на основі вже існуючих. Цей механізм забезпечує повторне використання коду, розширення функціональності та побудову ієрархій класів. У скриптових мовах програмування наслідування широко використовується для організації складних програмних систем.

Під наслідуванням розуміється здатність одного класу, який називається похідним або дочірнім, отримувати властивості і поведінку іншого класу, що називається базовим або батьківським. Похідний клас автоматично отримує доступ до атрибутів і методів базового класу, що дозволяє уникнути дублювання коду.

У скриптових мовах, таких як Python та JavaScript, наслідування реалізується за допомогою спеціальних конструкцій, що дозволяють визначити зв’язок між класами. Це дає змогу створювати ієрархії класів різного рівня складності.

Однією з ключових переваг наслідування є можливість розширення функціональності. Похідний клас може не лише використовувати можливості базового класу, але й додавати нові атрибути і методи. Це дозволяє адаптувати базову реалізацію до конкретних задач.

Перевизначення методів є важливим аспектом наслідування. Воно дозволяє змінювати поведінку методів, успадкованих від базового класу, у похідному класі. Це означає, що похідний клас може реалізувати власну версію методу, яка замінює або доповнює базову реалізацію.

При перевизначенні методу важливо забезпечити узгодженість інтерфейсу. Метод у похідному класі повинен мати ту ж сигнатуру, що і у базовому класі, щоб забезпечити коректну взаємодію між компонентами системи.

У деяких випадках похідний клас може викликати метод базового класу як частину своєї реалізації. Це дозволяє розширити поведінку, зберігаючи базову функціональність. Такий підхід часто використовується для поєднання загальної логіки з конкретними особливостями реалізації.

Поліморфізм є ще одним фундаментальним принципом об’єктно-орієнтованого програмування. Він передбачає можливість використання єдиного інтерфейсу для роботи з об’єктами різних типів. Це означає, що одна і та сама операція може виконуватися по-різному залежно від типу об’єкта.

Поліморфізм дозволяє створювати універсальні програмні рішення, які можуть працювати з різними типами даних без необхідності змінювати код. Це значно підвищує гнучкість і розширюваність системи.

У скриптових мовах поліморфізм часто реалізується на основі динамічної типізації. Наприклад, функція може працювати з різними типами об’єктів, якщо вони підтримують необхідні операції. Це дозволяє реалізувати так звану "качину типізацію", коли важливим є не тип об’єкта, а його поведінка.

Робота з колекціями даних є важливою частиною програмування, оскільки більшість задач передбачає обробку наборів значень. Колекції дозволяють зберігати і обробляти множини елементів, забезпечуючи зручні засоби доступу і маніпуляції даними.

До основних типів колекцій належать списки, масиви, множини та словники. Кожен з цих типів має свої особливості і призначений для вирішення певного класу задач. Наприклад, списки використовуються для зберігання впорядкованих даних, а словники – для зберігання пар ключ-значення.

У Python існує широкий набір вбудованих структур даних, що дозволяють ефективно працювати з колекціями. У JavaScript також підтримуються різні типи колекцій, включаючи масиви та об’єкти.

Операції над колекціями включають додавання, видалення, пошук і перебір елементів. Для цього використовуються як спеціальні методи, так і цикли. Ітерація по колекціях дозволяє обробляти кожен елемент послідовно.

Важливим аспектом є ефективність роботи з колекціями. Вибір типу колекції і способу обробки даних може суттєво впливати на продуктивність програми. Наприклад, операції пошуку у словниках зазвичай виконуються швидше, ніж у списках.

Поліморфізм також проявляється при роботі з колекціями. Одна і та сама операція, наприклад перебір елементів, може застосовуватися до різних типів колекцій. Це дозволяє створювати універсальні алгоритми обробки даних.

Таким чином, наслідування, перевизначення методів і поліморфізм є основою об’єктно-орієнтованого програмування у скриптових мовах. Вони дозволяють створювати

гнучкі, розширювані та повторно використовувані програмні компоненти. Робота з колекціями даних забезпечує ефективну обробку інформації і є невід'ємною частиною більшості програмних систем.

Лекція 14. Колекції, списки, кортежі, множини та словники

Колекції даних є одним із фундаментальних елементів програмування, оскільки вони дозволяють зберігати, організовувати та обробляти множини значень. У скриптових мовах програмування колекції відіграють ключову роль, адже більшість задач пов'язана з обробкою наборів даних різної структури. Використання відповідних типів колекцій дозволяє ефективно реалізовувати алгоритми та оптимізувати використання ресурсів.

У скриптових мовах, таких як Python та JavaScript, існує кілька основних типів колекцій, серед яких особливе місце займають списки, кортежі, множини та словники. Кожен із цих типів має свої властивості і використовується для розв'язання певного класу задач.

Списки є впорядкованими змінюваними колекціями, які дозволяють зберігати елементи різних типів. Вони підтримують доступ за індексом, що дає змогу швидко отримувати або змінювати значення. Списки широко використовуються для реалізації послідовностей даних, таких як набори чисел або рядків.

Кортежі є впорядкованими, але незмінними колекціями. Після створення кортежа його елементи не можуть бути змінені. Це робить кортежі більш безпечними з точки зору збереження даних, оскільки вони гарантують незмінність структури. Кортежі часто використовуються для представлення фіксованих наборів значень.

Множини є невпорядкованими колекціями унікальних елементів. Вони автоматично виключають дублікати і дозволяють виконувати операції над множинами, такі як об'єднання, перетин та різниця. Множини ефективно використовуються у задачах, де важлива перевірка належності елемента або робота з унікальними значеннями.

Словники є колекціями пар ключ-значення. Кожен елемент словника складається з ключа і відповідного значення. Ключі повинні бути унікальними, що забезпечує швидкий доступ до значень. Словники широко застосовуються для зберігання структурованих даних і реалізації асоціативних масивів.

Основні операції над колекціями включають додавання, видалення, пошук і зміну елементів. У списках можна додавати нові елементи, змінювати існуючі та видаляти непотрібні. У множинах доступні операції над множинами, що дозволяють виконувати математичні операції над наборами даних. У словниках можна додавати нові пари ключ-значення або змінювати значення за існуючим ключем.

Пошук елементів є важливою операцією при роботі з колекціями. У списках пошук може виконуватися шляхом послідовного перегляду елементів, тоді як у словниках і множинах використовуються більш ефективні механізми доступу, що забезпечують швидке виконання операцій.

Ітерація по структурах даних є одним із основних способів обробки колекцій. Вона дозволяє послідовно перебирати всі елементи колекції і виконувати певні дії для кожного з них. Ітерація реалізується за допомогою циклів, що забезпечують зручний доступ до елементів.

У Python ітерація по колекціях реалізується через універсальний механізм, який дозволяє працювати з різними типами даних однаковим чином. У JavaScript також існують спеціалізовані конструкції для перебору елементів масивів і об’єктів.

Ітерація може здійснюватися як за значенням, так і за індексом. У випадку списків часто використовується доступ за індексом, що дозволяє отримувати позицію елемента. У словниках і множинах ітерація зазвичай здійснюється по ключах або значеннях.

Важливим аспектом є ефективність операцій над колекціями. Різні типи колекцій мають різну складність виконання операцій, тому вибір відповідної структури даних є важливим для оптимізації програм. Наприклад, доступ до елемента словника за ключем виконується значно швидше, ніж пошук у списку.

У сучасних скриптових мовах також підтримуються розширені можливості роботи з колекціями, такі як генерація нових колекцій на основі існуючих. Це дозволяє компактно записувати складні операції обробки даних.

Колекції можуть бути вкладеними, тобто містити інші колекції як свої елементи. Це дозволяє створювати складні структури даних, що відображають реальні об’єкти і зв’язки між ними. Наприклад, словник може містити списки або інші словники.

Правильне використання колекцій дозволяє значно спростити реалізацію алгоритмів і підвищити ефективність програм. Вибір типу колекції залежить від характеру задачі, обсягу даних і вимог до продуктивності.

Таким чином, списки, кортежі, множини та словники є основними інструментами роботи з даними у скриптових мовах програмування. Вони забезпечують гнучкість, ефективність і зручність обробки інформації. Розуміння їх властивостей і операцій над ними є необхідною умовою для ефективного створення програмного забезпечення.

Лекція 15. Введення та виведення даних

Введення та виведення даних є невід’ємною складовою будь-якої програмної системи, оскільки забезпечує взаємодію програми з користувачем, файлами та зовнішніми джерелами інформації. У скриптових мовах програмування ці операції реалізуються за допомогою вбудованих засобів, що дозволяють ефективно організувати обробку як текстових, так і числових даних.

Введення даних у програму може здійснюватися різними способами залежно від задачі. Найпростішим є введення з клавіатури, коли користувач вводить дані під час виконання програми. У цьому випадку використовуються спеціальні функції, які зчитують введене значення і зберігають його у змінній. Отримані дані зазвичай мають текстовий формат і потребують подальшого перетворення у відповідний тип.

У скриптових мовах, таких як Python та JavaScript, введення даних може також здійснюватися з використанням файлів, мережевих запитів або інших джерел. Це дозволяє створювати програми, що працюють з великими обсягами інформації та автоматизують процес обробки даних.

Виведення результатів є процесом відображення обробленої інформації. Найчастіше результати виводяться на екран у вигляді тексту. Для цього використовуються спеціальні функції, які дозволяють форматувати дані і представляти їх у зручному для користувача вигляді. Важливим аспектом є зрозумілість і структурованість виводу.

Окрім виведення на екран, результати можуть записуватися у файли або передаватися іншим програмам. Це особливо актуально для задач, пов’язаних з обробкою великих обсягів даних або створенням звітів.

Робота з файлами є одним із основних способів збереження та обробки інформації. Файли дозволяють зберігати дані між запусками програми і забезпечують доступ до великих обсягів інформації. У скриптових мовах передбачені засоби для відкриття, читання, запису і закриття файлів.

Процес роботи з файлом зазвичай включає кілька етапів: відкриття файлу, виконання операцій читання або запису і закриття файлу. Відкриття файлу здійснюється у певному режимі, який визначає, чи буде файл використовуватися для читання, запису або додавання даних.

Зчитування даних з файлу може виконуватися построково або блоками. Построкове зчитування дозволяє обробляти великі файли без завантаження їх повністю у пам’ять, що є важливим для ефективності програм. Запис даних у файл здійснюється шляхом передачі інформації у відповідний потік.

При роботі з файлами важливо враховувати можливі помилки, такі як відсутність файлу, обмеження доступу або пошкодження даних. Для цього використовуються механізми обробки виняткових ситуацій, що дозволяють забезпечити надійність програм.

Обробка текстових даних є поширеною задачею у програмуванні. Вона включає операції над рядками, такі як пошук підрядків, заміна символів, розбиття рядків на частини та об’єднання даних. Скриптові мови надають широкий набір вбудованих функцій для роботи з текстом.

Числові дані також потребують спеціальної обробки. Це може включати виконання арифметичних операцій, обчислення статистичних показників або перетворення типів даних. Важливо враховувати точність обчислень і коректність результатів.

У прикладних задачах часто виникає необхідність обробки комбінованих даних, що містять як текстову, так і числову інформацію. У таких випадках застосовуються комплексні підходи, що поєднують різні методи обробки.

Основні методи обробки даних включають фільтрацію, сортування, агрегування та перетворення. Фільтрація дозволяє відбирати дані за певними критеріями. Сортування забезпечує впорядкування даних за визначеними ознаками. Агрегування дозволяє обчислювати узагальнені показники, такі як сума або середнє значення.

Перетворення даних передбачає зміну їх структури або формату. Наприклад, текстові дані можуть бути перетворені у числові, або навпаки. Це є необхідним для забезпечення сумісності даних з різними операціями.

Важливим аспектом є ефективність обробки даних. При роботі з великими обсягами інформації необхідно оптимізувати алгоритми і використовувати ефективні структури даних. Це дозволяє зменшити час виконання програми і використання ресурсів.

У скриптових мовах також підтримується робота з різними форматами файлів, такими як текстові, табличні або структуровані формати даних. Це розширює можливості програм і дозволяє інтегрувати їх з іншими системами.

Таким чином, введення та виведення даних, робота з файлами і обробка інформації є ключовими аспектами програмування у скриптових мовах. Вони забезпечують взаємодію програм з зовнішнім середовищем і дозволяють реалізовувати прикладні задачі різної складності. Розуміння цих механізмів є необхідною умовою для ефективної розробки програмного забезпечення.

Лекція 16. Обробка виняткових ситуацій

Обробка виняткових ситуацій є важливою складовою розробки надійного програмного забезпечення, оскільки дозволяє передбачати, виявляти і коректно обробляти помилки, що виникають під час виконання програми. У скриптових мовах програмування механізми обробки винятків забезпечують можливість контролювати поведінку програми у випадках, коли стандартне виконання не може бути продовжене.

Під винятковою ситуацією розуміється подія, що порушує нормальний хід виконання програми. Це може бути, наприклад, помилка доступу до файлу, некоректне введення даних або спроба виконання недопустимої операції. Без належної обробки такі ситуації можуть призвести до аварійного завершення програми.

У скриптових мовах, таких як Python та JavaScript, винятки представлені у вигляді об’єктів, що містять інформацію про помилку. Це дозволяє не лише виявити факт помилки, але й отримати детальну інформацію про її причини.

Типи винятків визначають класи помилок, які можуть виникати під час виконання програми. Наприклад, можуть існувати винятки, пов’язані з арифметичними операціями, доступом до файлів, роботою з пам’яттю або некоректними значеннями змінних. Класифікація винятків дозволяє більш точно реагувати на різні типи помилок.

У Python існує ієрархія винятків, що дозволяє групувати їх за типами і обробляти як окремі винятки, так і їх групи. У JavaScript також використовуються об’єкти помилок, що представляють різні типи виняткових ситуацій.

Конструкції обробки винятків дозволяють перехоплювати помилки і визначати альтернативні дії у випадку їх виникнення. Основною ідеєю є відокремлення основної логіки програми від логіки обробки помилок. Це підвищує читабельність коду і спрощує його супроводження.

Типова конструкція обробки винятків включає блок, у якому виконується основний код, і блоки, що обробляють помилки. Якщо під час виконання виникає виняток, керування передається відповідному обробнику, який визначає подальші дії.

У скриптових мовах обробка винятків дозволяє обробляти як конкретні типи помилок, так і всі винятки загалом. Це дає змогу гнучко реагувати на різні ситуації і забезпечувати стабільність роботи програми.

Додатково можуть використовуватися конструкції, що виконуються незалежно від того, чи виникла помилка. Вони застосовуються для звільнення ресурсів, закриття файлів або виконання інших обов’язкових дій. Це є важливим для забезпечення коректного завершення роботи програми.

Генерація винятків є механізмом, що дозволяє програмісту явно створювати виняткові ситуації у коді. Це використовується для сигналізації про помилки або некоректні стани програми. Наприклад, якщо функція отримує некоректні дані, вона може згенерувати виняток для повідомлення про проблему.

Генерація винятків дозволяє централізувати обробку помилок і забезпечити єдиний підхід до їх обробки. Це підвищує надійність і передбачуваність поведінки програми.

Важливим аспектом є правильне визначення місць генерації винятків. Необхідно чітко визначити, які ситуації є винятковими і потребують обробки. Надмірне використання винятків може ускладнювати логіку програми, тоді як їх недостатнє використання може призвести до пропуску помилок.

Обробка винятків також пов’язана з логуванням помилок. Фіксація інформації про помилки дозволяє аналізувати їх причини і покращувати якість програмного забезпечення. Це є особливо важливим у великих системах, де помилки можуть виникати у різних компонентах.

У прикладних задачах обробка винятків використовується для забезпечення стійкості програм. Наприклад, при роботі з файлами необхідно передбачити ситуації, коли файл не існує або не має необхідних прав доступу. У таких випадках програма повинна обробити помилку і продовжити роботу або повідомити користувача.

Ефективна обробка винятків передбачає баланс між контролем помилок і складністю коду. Необхідно уникати як повного ігнорування винятків, так і надмірного ускладнення логіки їх обробки.

Таким чином, обробка виняткових ситуацій є важливим інструментом забезпечення надійності програмного забезпечення. Вона дозволяє контролювати поведінку програми у випадку помилок, забезпечувати коректне завершення роботи і підвищувати якість програмних систем. Розуміння типів винятків, конструкцій їх обробки і механізмів генерації є необхідною умовою для ефективної роботи програміста.

Лекція 17. Забезпечення надійності програм

Надійність програмного забезпечення є однією з ключових характеристик якості, оскільки вона визначає здатність системи коректно виконувати свої функції в заданих умовах протягом визначеного часу. У скриптових мовах програмування, де часто використовується динамічна типізація і швидка ітеративна розробка, питання забезпечення надійності набуває особливої актуальності.

Під надійністю програм розуміється здатність програмного забезпечення функціонувати без збоїв, обробляти помилки і забезпечувати передбачувану поведінку навіть у випадку некоректних вхідних даних або несприятливих умов виконання. Досягнення цього вимагає системного підходу до проєктування, реалізації та тестування програм.

Одним із основних способів підвищення надійності є перевірка вхідних даних. Програма повинна контролювати коректність значень, що надходять від користувача або з інших джерел, і обробляти некоректні дані відповідним чином. Це дозволяє запобігти виникненню помилок під час виконання.

Важливу роль відіграє обробка виняткових ситуацій, яка дозволяє контролювати помилки і забезпечувати коректне завершення роботи програми. Використання механізмів обробки винятків дозволяє ізолювати проблемні ділянки коду і мінімізувати їх вплив на загальну роботу системи.

Налагодження програмного коду є процесом виявлення, аналізу та усунення помилок у програмі. Воно є невід’ємною частиною процесу розробки і виконується на різних етапах створення програмного забезпечення. У скриптових мовах налагодження часто виконується інтерактивно, що дозволяє швидко перевіряти зміни у коді.

Помилки у програмі можуть мати різну природу. Синтаксичні помилки виникають через порушення правил запису коду і зазвичай виявляються інтерпретатором до початку виконання. Логічні помилки пов’язані з некоректною реалізацією алгоритму і можуть бути складнішими для виявлення. Помилки виконання виникають під час роботи програми і часто пов’язані з некоректними даними або станом системи.

Методи пошуку помилок включають аналіз коду, використання тестів і поступове виконання програми. Аналіз коду дозволяє виявити очевидні помилки у логіці або структурі програми. Тестування дозволяє перевірити правильність роботи програми на різних наборах даних.

Одним із ефективних методів є трасування виконання програми. Це передбачає послідовний аналіз виконання інструкцій і відстеження значень змінних. Такий підхід дозволяє виявити місце виникнення помилки і визначити її причину.

Використання виведення проміжних результатів є простим, але ефективним способом налагодження. Додавання додаткових повідомлень у код дозволяє контролювати хід виконання програми і перевіряти значення змінних у різних точках.

Засоби відлагодження є спеціалізованими інструментами, що дозволяють автоматизувати процес налагодження. У середовищах розробки передбачені можливості встановлення точок зупинки, покрокового виконання коду, перегляду значень змінних і аналізу стану програми.

У скриптових мовах, таких як Python та JavaScript, існують вбудовані засоби відлагодження, що дозволяють ефективно працювати з програмним кодом. Вони забезпечують можливість детального аналізу виконання і швидкого виявлення помилок.

Покрокове виконання дозволяє виконувати програму по одній інструкції і спостерігати за змінами стану системи. Це є особливо корисним при аналізі складних алгоритмів або пошуку логічних помилок.

Точки зупинки дозволяють зупинити виконання програми у визначених місцях і дослідити стан змінних. Це дає змогу зосередитися на конкретних ділянках коду і більш детально проаналізувати їх роботу.

Перегляд стеку викликів дозволяє визначити послідовність викликів функцій, що призвели до виникнення помилки. Це є важливим для розуміння контексту виконання і пошуку причин проблеми.

Виправлення помилок є завершальним етапом налагодження. Воно передбачає внесення змін у код і повторну перевірку його роботи. Важливо переконатися, що виправлення не призвело до появи нових помилок.

Тестування після внесення змін є обов’язковим етапом, що дозволяє перевірити коректність роботи програми. Використання автоматизованих тестів дозволяє швидко перевіряти великі обсяги коду і забезпечує стабільність системи.

Забезпечення надійності програм також включає документування і аналіз помилок. Фіксація проблем і способів їх вирішення дозволяє накопичувати досвід і покращувати процес розробки.

Таким чином, забезпечення надійності програмного забезпечення, налагодження коду і використання методів пошуку та виправлення помилок є важливими складовими професійної діяльності інженера програмного забезпечення. Використання сучасних засобів відлагодження дозволяє значно підвищити ефективність розробки і забезпечити високу якість програмних систем.

Лекція 18. Тестування програмного забезпечення

Логування подій є важливим інструментом забезпечення надійності та керованості програмного забезпечення, оскільки воно дозволяє фіксувати інформацію про хід виконання програми, виявляти помилки та аналізувати поведінку системи у різних умовах. У

скриптових мовах програмування механізми логування широко використовуються як під час розробки, так і на етапі експлуатації програм.

Під логуванням розуміється процес запису повідомлень про події, що відбуваються у програмі. Ці повідомлення можуть містити інформацію про виконання операцій, зміну стану системи, виникнення помилок або інші важливі події. Логування дозволяє відстежувати роботу програми і аналізувати її поведінку без необхідності прямого втручання у процес виконання.

Логування може здійснюватися на різних рівнях деталізації. Найбільш поширеними є інформаційні повідомлення, попередження та повідомлення про помилки. Така класифікація дозволяє фільтрувати повідомлення і зосереджувати увагу на найбільш важливих подіях.

У скриптових мовах, таких як Python та JavaScript, існують вбудовані засоби для організації логування, що дозволяють налаштовувати формат повідомлень, рівні логування та місце збереження логів. Це може бути консоль, файл або зовнішня система.

Використання логування є більш ефективним підходом, ніж просте виведення повідомлень у консоль, оскільки воно дозволяє централізовано керувати повідомленнями і зберігати історію подій. Це є особливо важливим для аналізу помилок у складних системах.

Тестування програмного забезпечення є процесом перевірки відповідності програми заданим вимогам і виявлення дефектів. Воно є невід'ємною частиною життєвого циклу програмного забезпечення і забезпечує його якість та надійність.

Метою тестування є не лише виявлення помилок, але й підтвердження того, що програма працює відповідно до очікувань. Тестування дозволяє перевірити коректність реалізації алгоритмів, обробку даних і взаємодію між компонентами системи.

Методи тестування програм можуть бути класифіковані за різними ознаками. Однією з основних класифікацій є поділ на тестування типу «чорної скриньки» і «білої скриньки». У першому випадку перевіряється поведінка програми без аналізу її внутрішньої структури, а у другому – враховується структура коду.

Тестування типу «чорної скриньки» орієнтоване на перевірку функціональності програми. Воно дозволяє оцінити, чи відповідають результати роботи програми заданим вимогам. Для цього використовуються різні набори вхідних даних, включаючи граничні значення і некоректні дані.

Тестування типу «білої скриньки» передбачає аналіз внутрішньої структури програми і перевірку різних шляхів виконання коду. Це дозволяє виявити помилки у логіці реалізації і забезпечити покриття коду тестами.

Існують також інші види тестування, такі як модульне, інтеграційне та системне. Модульне тестування передбачає перевірку окремих компонентів програми, інтеграційне – перевірку взаємодії між ними, а системне – тестування всієї системи в цілому.

У скриптових мовах широко використовуються автоматизовані засоби тестування, що дозволяють створювати тестові сценарії і виконувати їх автоматично. Це значно підвищує ефективність тестування і дозволяє швидко виявляти помилки при внесенні змін у код.

Перевірка коректності результатів є важливим етапом тестування. Вона передбачає порівняння фактичних результатів виконання програми з очікуваними. Для цього використовуються тестові випадки, що містять вхідні дані і очікувані результати.

Особливу увагу слід приділяти перевірці граничних умов і нестандартних ситуацій. Саме у таких випадках найчастіше виникають помилки. Тому тестування повинно охоплювати як типові, так і екстремальні сценарії використання програми.

У прикладних задачах перевірка коректності може включати аналіз статистичних показників, перевірку відповідності форматів даних або контроль логічних умов. Це дозволяє забезпечити достовірність отриманих результатів.

Важливим аспектом є повторюваність тестування. Тестові сценарії повинні бути сформульовані таким чином, щоб їх можна було виконувати багаторазово з однаковими результатами. Це забезпечує об'єктивність оцінки якості програмного забезпечення.

Логування і тестування часто використовуються разом. Логи дозволяють отримати додаткову інформацію про виконання тестів і виявити причини помилок. Це спрощує процес налагодження і підвищує ефективність розробки.

Таким чином, логування подій, тестування програмного забезпечення і перевірка коректності результатів є ключовими інструментами забезпечення якості програм. Вони дозволяють виявляти помилки, аналізувати поведінку системи і гарантувати відповідність програм заданим вимогам. Розуміння цих процесів є необхідною умовою для ефективної роботи інженера програмного забезпечення.

Лекція 19. Розробка прикладних програм

Розробка прикладних програм є основною сферою застосування скриптових мов програмування, оскільки вони забезпечують швидке створення програмних рішень для розв'язання практичних задач. Прикладні програми орієнтовані на кінцевого користувача і виконують конкретні функції, такі як обробка даних, автоматизація процесів або взаємодія з інформаційними системами.

Процес розробки прикладної програми починається з аналізу задачі і формулювання вимог. Необхідно визначити, які функції повинна виконувати програма, які дані вона обробляє і які результати має формувати. Це дозволяє створити чітке уявлення про структуру майбутньої системи.

Структура програмного проєкту визначає організацію файлів, модулів і пакетів, що входять до складу програми. Вона повинна бути логічною, зрозумілою і зручною для супроводження. У скриптових мовах програмування структура проєкту зазвичай включає основний файл запуску, модулі з бізнес-логікою, допоміжні компоненти та ресурси.

Основний файл програми відповідає за ініціалізацію системи і запуск виконання. Він може містити виклики основних функцій або створення об'єктів, що забезпечують роботу програми. Інші модулі реалізують окремі частини функціональності, такі як обробка даних, взаємодія з файлами або реалізація алгоритмів.

У скриптових мовах, таких як Python та JavaScript, проєкти часто організовуються у вигляді ієрархії каталогів, що відповідає логічній структурі програми. Це дозволяє розділити код на окремі компоненти і спростити навігацію.

Важливим аспектом є розділення коду на рівні відповідальності. Наприклад, окремо виділяється логіка обробки даних, інтерфейс користувача і доступ до зовнішніх ресурсів. Такий підхід дозволяє зменшити залежності між компонентами і підвищити гнучкість системи.

Використання бібліотек є невід'ємною частиною розробки прикладних програм. Бібліотеки дозволяють реалізувати складні функції без необхідності їх розробки з нуля. Це значно прискорює процес створення програм і підвищує їх якість.

У стандартних бібліотеках скриптових мов містяться засоби для роботи з файлами, мережею, математичними обчисленнями та іншими задачами. Сторонні бібліотеки

розширюють ці можливості і дозволяють працювати з більш складними системами, такими як бази даних або вебсервіси.

Програмні інтерфейси, або API, є засобом взаємодії між різними компонентами програмного забезпечення. Вони визначають набір функцій і правил, за допомогою яких одна програма може використовувати можливості іншої. Використання API дозволяє інтегрувати різні системи і створювати складні програмні рішення.

У скриптових мовах програмні інтерфейси широко використовуються для роботи з вебсервісами, базами даних і зовнішніми ресурсами. Наприклад, програма може отримувати дані з віддаленого сервера або передавати інформацію іншим системам.

Важливим аспектом є правильна організація взаємодії з бібліотеками і API. Необхідно враховувати вимоги до форматів даних, протоколів обміну і обробки помилок. Це дозволяє забезпечити стабільну і коректну роботу програми.

Керування залежностями також відіграє важливу роль у структурі проєкту. Необхідно чітко визначити, які бібліотеки використовуються, і контролювати їх версії. Це забезпечує відтворюваність середовища і стабільність роботи програми.

Тестування прикладної програми є обов'язковим етапом розробки. Воно дозволяє перевірити коректність роботи всіх компонентів і їх взаємодії. Особливу увагу слід приділяти тестуванню інтеграції з бібліотеками і API.

Документування проєкту є важливим для його подальшого супроводження. Воно включає опис структури програми, використаних бібліотек і принципів роботи системи. Це дозволяє іншим розробникам швидко зрозуміти логіку програми і продовжити її розвиток.

У сучасних умовах розробка прикладних програм часто передбачає використання систем контролю версій, що дозволяють відстежувати зміни у коді і організовувати командну роботу. Це є важливим для забезпечення якості і стабільності програмного забезпечення.

Таким чином, розробка прикладних програм у скриптових мовах базується на правильній організації структури проєкту, ефективному використанні бібліотек і програмних інтерфейсів. Це дозволяє створювати гнучкі, масштабовані та надійні програмні системи, що відповідають сучасним вимогам.

Лекція 20. Робота з API

Робота з програмними інтерфейсами застосування є важливим напрямом сучасного програмування, оскільки дозволяє інтегрувати різні програмні системи, обмінюватися даними та розширювати функціональність прикладних програм. API визначає набір правил і механізмів взаємодії між програмними компонентами, що дозволяє використовувати зовнішні сервіси без необхідності реалізації їх внутрішньої логіки.

У скриптових мовах програмування, таких як Python та JavaScript, робота з API зазвичай здійснюється через мережеві запити. Програма формує запит до сервера, отримує відповідь у певному форматі та обробляє її відповідно до поставленої задачі. Найчастіше використовуються текстові формати обміну даними, що дозволяє легко інтегрувати різні системи.

Інтеграція програмних компонентів передбачає об'єднання різних частин програмного забезпечення в єдину систему. Це може включати взаємодію між модулями, бібліотеками, сервісами або зовнішніми ресурсами. Основною метою інтеграції є забезпечення узгодженої роботи всіх компонентів.

Важливим аспектом інтеграції є визначення інтерфейсів взаємодії. Кожен компонент повинен мати чітко визначений набір функцій і правил обміну даними. Це дозволяє зменшити залежності і забезпечити гнучкість системи.

Автоматизація задач за допомогою скриптів є одним із основних напрямів використання скриптових мов. Скрипти дозволяють виконувати повторювані операції без участі користувача, що значно підвищує ефективність роботи. Це може включати обробку файлів, взаємодію з системними ресурсами або виконання мережевих запитів.

Автоматизація дозволяє зменшити кількість ручних операцій, знизити ймовірність помилок і прискорити виконання задач. Скрипти можуть запускатися за розкладом або у відповідь на певні події, що забезпечує гнучкість їх використання.

Обробка даних у прикладних задачах часто включає зчитування інформації з різних джерел, її аналіз і формування результатів. Скриптові мови надають широкий набір засобів для роботи з даними, що дозволяє реалізувати складні алгоритми обробки.

У процесі обробки даних можуть використовуватися різні методи, такі як фільтрація, сортування, агрегування і перетворення. Це дозволяє отримати необхідну інформацію і представити її у зручному вигляді.

Створення утиліт для автоматизації процесів є важливим напрямом практичного застосування скриптових мов. Утиліти можуть виконувати допоміжні функції, такі як обробка файлів, моніторинг системи або інтеграція з іншими програмами. Вони часто мають невеликий обсяг, але виконують важливі задачі.

При розробці утиліт важливо забезпечити їх універсальність і зручність використання. Це досягається за рахунок параметризації, обробки помилок і документування функціональності. Утиліти повинні бути зрозумілими і легко інтегруватися у існуючі процеси.

Організація програмного коду є важливим аспектом розробки, що впливає на його читабельність і супроводжуваність. Код повинен бути структурований, логічно розподілений на модулі і відповідати принципам модульності та інкапсуляції.

Використання зрозумілих імен змінних і функцій, дотримання єдиного стилю оформлення і наявність коментарів дозволяють значно покращити якість коду. Це є особливо важливим у командній розробці, де код використовується різними розробниками.

Документування програмного коду є необхідною умовою його ефективного використання і супроводження. Документація може включати опис функцій, параметрів, форматів даних і принципів роботи програми. Це дозволяє швидко зрозуміти логіку системи і спростити її розвиток.

У скриптових мовах часто використовуються вбудовані механізми документування, що дозволяють створювати опис функцій безпосередньо у коді. Це забезпечує актуальність документації і її відповідність реалізації.

Важливим аспектом є підтримка актуальності документації. Зміни у коді повинні супроводжуватися відповідними змінами у документації, щоб уникнути розбіжностей між описом і реалізацією.

Таким чином, робота з API, інтеграція компонентів, автоматизація задач і організація коду є ключовими аспектами сучасного програмування у скриптових мовах. Вони дозволяють створювати ефективні, гнучкі та надійні програмні рішення, що відповідають вимогам практичних задач.

Рекомендована література

Основна

1. Педяш В. В., Йона Л. Г., Мазур Г. Д., Русу О. П. Python-програмування: методичні рекомендації для практичних та лабораторних занять [Електронне видання] / кафедра комп'ютерної інженерії та інноваційних технологій Міжнародного гуманітарного університету. Одеса, 2025. 137 с.

2. Костюченко А. О. Основи програмування мовою Python: навчальний посібник. Черкаси: ФОП Баликіна С. М., 2020. 180 с.

3. Яковенко А. В. Основи програмування. Python. Частина 1: підручник для студентів спеціальності 122 Комп'ютерні науки, спеціалізації Інформаційні технології в біології та медицині. Київ: КПІ ім. Ігоря Сікорського, 2018. 195 с.

4. Маттес Е. Пришвидшений курс Python. Київ: Видавництво Старого Лева, 2021. 600 с.

5. Васильєв О. Програмування мовою Python. Тернопіль: Навчальна книга – Богдан, 2019. 204 с.

6. Ceder N. The Quick Python Book. 3rd ed. New York: Manning Publications, 2018. 432 p.

7. Lambert K. A. Fundamentals of Python: First Programs. Boston: Cengage Learning, 2018. 476 p.

8. Lutz M. Learning Python. 5th ed. Sebastopol: O'Reilly Media, 2019. 648 p.

9. Стрелковська І. В., Григор'єва Т. І., Педяш В. В., Русу О. П., Мельник В. В. Використання Python програмування у графових моделях безпеки баз даних. Наука і техніка сьогодні. 2026. № 1 (55). С. 2570–2580. DOI: 10.52058/2786-6025-2026-1(55)-2570-2580.

Допоміжна

10. Fenner M. Machine Learning with Python for Everyone. Addison-Wesley Data & Analytics Series. Boston: Addison-Wesley Professional, 2019. 586 p.

11. Григор'єва Т., Русу О., Горбачов В., Крохмаль М. Багатовимірний підхід в задачах розробки програмного забезпечення. Вісник Хмельницького національного університету. Серія: Технічні науки. 2025. Т. 359, № 6.2. С. 157–160. DOI: https://doi.org/10.31891/2307-5732-2025-359-92

.

Інформаційні ресурси

12. Бібліотека Міжнародного гуманітарного університету: веб-сайт. URL: https://mu.od.ua/naukova-biblioteka

13. Національна бібліотека України ім. В. І. Вернадського: веб-сайт. URL: http://www.nbuv.gov.ua

Навчальне видання

Педяш Володимир Віталійович

СКРИПТОВІ МОВИ ПРОГРАМУВАННЯ

Опорний конспект лекцій для здобувачів вищої освіти,
які навчаються за спеціальностями галузі «Інформаційні технології»

Українською мовою