МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

Національний технічний університет
„Харківський політехнічний інститут ”

ПРОГРАМУВАННЯ ТА АЛГОРИТМІЗАЦІЯ

Конспект лекцій
для студентів спеціальності 172 «Електронні комунікації та радіотехніка» та 122 «Комп’ютерні науки»
денної та заочної форм навчання

Затверджено
редакційно-видавничою радою
університету,
протокол №2 від 26.06.2025

Харків
НТУ “ХПІ”
2025

Конспект лекцій з дисципліни «Програмування та алгоритмізація» для студентів спеціальності 172 «Електронні комунікації та радіотехніка» та 122 «Комп'ютерні науки» денної та заочної форм навчання / Склав М.В. Савченко. – Харків: НТУ "ХПІ", 2024. – 205 с.

Упорядник М.В. Савченко

Рецензент О.І. Трубаев

Кафедра системи інформації ім. В.О. Кравця

Вступ

Лекції, які увійшли до даного посібника, були прочитані доцентом Савченко Миколою Володимировичем восени 2024 року для студентів 2-го курсу кафедри Системи інформації. Відбір матеріалів до цього курсу у великій мірі був пов'язаний з назвою останнього, оскільки потрібно було познайомити з величезним масивом інформації, яка стосується використанню програмування в інформаційних технологіях. З курсом у більш повному об’ємі можна познайомитись на сайті курсу за адресою:

https://classroom.google.com/c/NjI4ODM4NTkwODNa?cjc=rw5pem5

.

Лекція №1. Вступ до програмування та Python

1.1 Визначення програмування та його роль у сучасному світі

Програмування – це процес створення програмного коду для виконання певних завдань комп'ютером. Це мистецтво та наука одночасно, що вимагає логічного мислення, творчості та вміння розуміти проблеми і шукати їхні розв'язки через програми.

Роль програмування у сучасному світі є надзвичайно важливою і впливає на різні сфери нашого життя. Ось деякі аспекти ролі програмування:

1. Технологічний розвиток: Програмування є головним драйвером технологічного розвитку. Всі інновації у сфері програмного забезпечення, мобільних технологій, штучного інтелекту, розумних пристроїв і т.д. базуються на програмуванні.

2. Ефективність та автоматизація: Програмування дозволяє автоматизувати багато рутинних процесів і операцій, що підвищує ефективність роботи у різних галузях, включаючи виробництво, логістику, банківську справу, медицину тощо.

3. Розваги і розваги: Багато з найпопулярніших розважальних продуктів, таких як відеоігри, мультимедійні додатки, стрімінгові платформи, музичні сервіси, розробляються за допомогою програмування.

4. Наука і дослідження: У багатьох наукових галузях, таких як фізика, біологія, хімія, астрономія, програмування використовується для аналізу даних, моделювання складних систем та проведення експериментів.

5. Інновації та стартапи: Багато стартапів і новаторських компаній засновані на ідеях, що базуються на програмному забезпеченні. Програмування дозволяє втілювати нові ідеї в життя та створювати інноваційні продукти і сервіси.

6. Комунікація та взаємодія: Веб-розробка і соціальні мережі, такі як Facebook, Twitter, Instagram, спрощують комунікацію і взаємодію між людьми навіть на великій відстані.

Узагальнюючи, програмування є важливою складовою сучасного світу, що визначає наші можливості, способи взаємодії та розвиток в різних

сферах життя. Воно впливає на всі аспекти нашого повсякденного і професійного життя, і важливою властивістю для тих, хто хоче бути ефективним у цифровому суспільстві.

1.2 Огляд мов програмування

Сучасний ландшафт мов програмування дуже різноманітний, і кожна мова має свої особливості, призначення та переваги. Нижче наведений огляд деяких з найбільш популярних мов програмування, які активно використовуються в сучасному програмуванні:

1. Python:
 – Особливості: Простий синтаксис, високорівнева мова, динамічна типізація, об'єктно-орієнтоване та функціональне програмування, широке застосування у веб-розробці, наукових дослідженнях, аналізі даних, штучному інтелекті та ін.

2. Java:
 – Особливості: Крос-платформена мова, об'єктно-орієнтоване програмування, висока надійність, широке застосування у розробці веб-додатків, мобільних додатків (Android), корпоративних системах.

3. JavaScript:
 – Особливості: Скриптова мова, широке застосування у веб-розробці (фронтенд і бекенд), можливість виконання на клієнтському та серверному боці, підтримка асинхронного програмування.

4. C#:
 – Особливості: Розроблена компанією Microsoft, об'єктно-орієнтоване програмування, широке застосування у розробці програм для платформи .NET, включаючи веб-додатки, мобільні додатки (Xamarin), ігри (Unity).

5. C++:
 – Особливості: Мова низькорівневого програмування, ефективне керування пам'яттю, широке застосування у вбудованому програмуванні, графічному програмуванні, розробці операційних систем та ігор.

6. Go (або Golang):
 – Особливості: Створена компанією Google, проста синтаксична структура, ефективне керування пам'яттю, підтримка паралельного програмування, широке застосування у веб-розробці, серверному програмуванні та розробці мікросервісів.

7. Ruby:
– Особливості: Простий та елегантний синтаксис, об'єктно-орієнтоване програмування, активне співтовариство (Ruby on Rails), підтримка функціонального програмування, широке застосування у веб-розробці.

8. Swift:
– Особливості: Розроблена компанією Apple, використовується для розробки програм для платформи iOS, об'єктно-орієнтоване та функціональне програмування, безпека типів даних.

Це лише декілька з найбільш популярних мов програмування в сучасному програмуванні. Кожна з них має свої сильні сторони та використовується для різноманітних цілей та завдань.

Сучасні мови програмування мають свої переваги, але вони також можуть мати недоліки. Ось деякі з них:

1. Швидкодія: Деякі високорівневі мови програмування, які забезпечують великий рівень абстракції, можуть бути менш ефективними з точки зору продуктивності порівняно з мовами низького рівня. Це може бути особливо важливим для додатків, що потребують великої швидкодії або низької латентності.

2. Споживання пам'яті: Деякі мови програмування можуть споживати більше пам'яті для виконання тих самих завдань порівняно з іншими мовами. Це може бути проблемою для додатків, які працюють з великими обсягами даних або мають обмежені ресурси пам'яті.

3. Синтаксичний сахар: Деякі мови програмування надають багато синтаксичного цукру та скорочень, що робить код коротшим і зрозумілішим. Однак це може призвести до того, що розуміння та налагодження коду стають складнішими для новачків або навіть для досвідчених розробників.

4. Сумісність: У багатьох мовах програмування може бути проблема з сумісністю між різними версіями або імплементаціями мови. Це може призвести до проблем з переносом коду між платформами або різними середовищами.

5. Відсутність документації: Деякі мови програмування можуть мати обмежену або неповну документацію, що робить навчання та розробку

програм складнішою. Добре документовані мови сприяють швидкому вивченню та розвитку програм.

6. Залежність від інструментів та бібліотек: Багато сучасних мов програмування покладаються на інструменти, бібліотеки та фреймворки для реалізації багатьох функцій та функціональностей. Це може створювати проблеми з управлінням залежностями та підтримкою коду на довгостроковому горизонті.

Це лише кілька недоліків, які можуть бути пов'язані з сучасними мовами програмування. Важливо враховувати ці аспекти під час вибору мови програмування для конкретного проекту або завдання.

1.3 Властивості мови Python

Мова програмування Python була розроблена Гвідо ван Россумом (Guido van Rossum). Вона була вперше випущена у 1991 році, і з того часу стала однією з найпопулярніших та широко використовуваних мов програмування у світі. Гвідо ван Россум придумав Python, коли працював у Нідерландах у групі програмістів, які працювали над розробкою операційної системи Amoeba. Він вирішив створити мову програмування, яка була б простою та зрозумілою для користувачів, а також має чіткий та лаконічний синтаксис.

Мова програмування Python має багато властивостей, які роблять її популярною серед розробників. Ось деякі з найбільш помітних властивостей Python:

1. Простота вивчення та читання коду: Python відомий своєю чіткістю та простотою синтаксису, що робить код легким у вивченні та розумінні, навіть для початківців.

2. Динамічна типізація: У Python не потрібно явно вказувати типи змінних, оскільки типи визначаються автоматично при присвоєнні значень. Це спрощує процес програмування та зменшує кількість коду.

3. Широкий вбудований функціонал: Python має велику стандартну бібліотеку, яка включає в себе різноманітні інструменти для роботи з рядками, файлами, мережами, базами даних, математичними обчисленнями та іншими завданнями.

4. Підтримка об'єктно-орієнтованого програмування (ООП): Python підтримує об'єктно-орієнтований підхід до програмування, що дозволяє розбити програму на невеликі, самодостатні блоки (об'єкти), які можуть взаємодіяти між собою.

5. Інтерпретованість: Python є інтерпретованою мовою, що означає, що код виконується рядок за рядком, без необхідності компіляції. Це полегшує тестування та налагодження програм.

6. Переносимість: Python є крос-платформеним, тобто програми, написані на Python, можуть запускатися на різних операційних системах без змін.

7. Автоматичне управління пам'яттю: Python використовує механізм автоматичного збирання сміття, що дозволяє автоматично вивільняти пам'ять, яку більше не використовує програма.

8. Широкий спектр застосувань: Python може бути використаний для розробки веб-додатків, наукових обчислень, штучного інтелекту, аналізу даних, автоматизації, ігор та багато іншого.

Ці властивості роблять Python потужним і універсальним інструментом для різноманітних завдань програмування.

Так, хоча мова програмування Python є дуже популярною і має багато переваг, вона також має свої недоліки. Ось деякі з них:

1. Швидкодія: Python, порівняно з деякими іншими мовами програмування, такими як C++ або Java, може бути менш ефективним з точки зору швидкодії. Це пов'язано з тим, що Python є інтерпретованою мовою, а не компілюється в машинний код. Хоча це не проблема для багатьох застосувань, в областях, де потрібна висока продуктивність, така як графічні ігри або високонавантажені веб-системи, це може бути обмеженням.

2. Глобальний інтерпретатор: У деяких випадках різниця між версіями Python (наприклад, Python 2 і Python 3) може створювати проблеми з сумісністю програм. Деякі бібліотеки або фреймворки можуть підтримувати тільки одну з версій, що ускладнює розвиток і підтримку коду.

3. Неявна типізація: Хоча динамічна типізація Python є перевагою з багатьох точок зору, вона також може створювати проблеми в більших проектах або для команд розробників, особливо коли код стає складнішим і важким для розуміння.

4. Обмежена підтримка мобільних додатків: Хоча Python широко використовується для розробки серверних та веб-додатків, він не так популярний для розробки мобільних додатків порівняно з мовами, такими як Java для Android або Swift для iOS. І хоча існують фреймворки, такі як Kivy, які дозволяють створювати мобільні додатки на Python, їхні можливості можуть бути обмеженими.

5. Неможливість приховати код: Оскільки Python є інтерпретованою мовою, код може бути легко доступним, що може бути проблемою для захисту інтелектуальної власності або конфіденційної інформації.

Хоча Python має свої недоліки, він все ще є однією з найбільш популярних мов програмування завдяки своїм перевагам у простоті використання, гнучкості та широкому спектру застосувань.

1.4 Встановлення та налаштування середовища для програмування на Python

Встановлення та налаштування середовища для програмування на Python досить просте. Ось кроки, які ви можете виконати:

1. Завантажте Python: Перш за все, вам потрібно завантажити та встановити Python. Ви можете зробити це, відвідавши веб-сайт Python за посиланням [python.org](https://www.python.org/), вибравши потрібну версію Python для своєї операційної системи та слідуючи інструкціям по встановленню.

2. Встановіть редактор коду: Ви можете використовувати будь-який текстовий редактор або інтегроване середовище розробки (IDE) для програмування на Python. Деякі популярні варіанти включають Visual Studio Code, PyCharm, Sublime Text, Atom тощо. Виберіть той, який вам найбільше подобається.

3. Перевірте встановлення Python: Після встановлення Python відкрийте термінал (у Windows це може бути командний рядок або PowerShell, у MacOS або Linux – термінал) і введіть `python --version` або `python3 --version` (залежно від того, як Python зберігається у вашій системі) щоб переконатися, що встановлення пройшло успішно.

4. Встановіть додаткові пакети (опціонально): Якщо ви плануєте використовувати додаткові бібліотеки або інструменти для розробки, встановіть їх за допомогою менеджера пакетів pip. Наприклад, для встановлення бібліотеки NumPy ви можете використовувати команду `pip install numpy`.

5. Створіть свій перший скрипт: Відкрийте свій обраний текстовий редактор або IDE та створіть новий файл з розширенням ".py". Напишіть простий Python код, наприклад:

```
print("Hello, world!")
```

Збережіть файл і запустіть його з терміналу, використовуючи команду `python ваш_файл.py`. Ви повинні побачити вивід "Hello, world!".

6. Налаштуйте IDE (опціонально): Якщо ви використовуєте IDE, зазвичай в ньому є можливість налаштування додаткових параметрів, наприклад, встановлення автодоповнення коду, інтеграцію з Git, встановлення правил оформлення коду тощо.

Це загальний огляд процесу встановлення та налаштування середовища для програмування на Python. Бажаю успіхів у вашому програмуванні!

Так, у Python є багато фреймворків для різноманітних цілей та галузей програмування. Ось деякі з найпопулярніших фреймворків Python:

1. Django: Django є одним з найпопулярніших веб-фреймворків Python. Він надає повний набір інструментів для швидкого розроблення веб-додатків, включаючи адміністративний інтерфейс, ORM (Object-Relational Mapping), систему маршрутизації та інше.

2. Flask: Flask є легковаговим веб-фреймворком Python, який надає базовий набір інструментів для розробки веб-додатків. Він має простий синтаксис і добре підходить для маленьких та середніх проектів.

3. Pyramid: Pyramid є фреймворком для веб-розробки Python, який надає гнучкість та широкі можливості розширення. Він підходить як для невеликих, так і для складних веб-додатків.

4. FastAPI: FastAPI є новим фреймворком Python для створення високопродуктивних веб-додатків та API. Він використовує сучасні асинхронні технології та надає автоматичну документацію Swagger.

5. PyQt / PySide: Це набори бібліотек для розробки десктопних додатків з використанням графічного інтерфейсу користувача (GUI). Вони базуються на Qt, крос-платформенному фреймворку для розробки програм.

6. TensorFlow / PyTorch: Ці фреймворки використовуються для розробки моделей машинного навчання та глибокого навчання. Вони надають різноманітні інструменти для створення, навчання та оцінки моделей нейронних мереж.

Це лише кілька прикладів фреймворків для розробки програм на мові Python. Ці фреймворки надають розробникам потужні інструменти для швидкої та ефективної розробки програм з різноманітними функціями та можливостями.

Питання для перевірки знань по темі лекції

1. Що таке програмування і яке його призначення у сучасному світі?
Програмування — це процес створення інструкцій для комп'ютера, які він виконує для вирішення певних завдань. Його основна мета — автоматизація рутинних завдань, обробка даних, керування пристроями, створення програмного забезпечення та розробка систем, які сприяють вирішенню складних завдань. У сучасному світі програмування є основою для розвитку інформаційних технологій, воно проникає в усі сфери життя, забезпечуючи ефективну роботу в науці, медицині, бізнесі, освіті та інших галузях.

2. Які є основні мови програмування, і як вони використовуються у різних галузях?
Основні мови програмування включають:
- Python: Використовується в наукових дослідженнях, машинному навчанні, веб-розробці, автоматизації та аналізі даних.
- JavaScript: Основна мова для веб-розробки, створення інтерфейсів користувача, а також для серверної сторони через Node.js.
- Java: Популярна в корпоративних рішеннях, мобільних додатках (особливо Android), а також у великих системах обробки даних.
- C++: Застосовується у розробці ігор, системного програмного забезпечення, вбудованих систем і додатків, що потребують високої продуктивності.
- C#: Використовується для розробки програмного забезпечення під Windows, ігор за допомогою Unity, а також у корпоративних додатках.
- SQL: Мова запитів до баз даних, використовується для управління та обробки великих обсягів даних.

3. Наведіть приклади сфер життя, де використання програмування має велике значення.
- Медицина: Програмне забезпечення для діагностики, управління електронними медичними записами, розробка систем штучного інтелекту для прогнозування захворювань.
- Фінанси: Автоматизація трейдингу, обробка великих даних для аналітики, забезпечення безпеки транзакцій.
- Освіта: Онлайн-навчання, створення навчальних платформ, адаптивні системи тестування.
- Розваги: Розробка відеоігор, віртуальної та доповненої реальності, створення медіаплатформ для потокового передавання контенту.

- Транспорт: Автоматизація логістики, системи навігації, розробка програмного забезпечення для самокерованих автомобілів.

4. Як програмування сприяє автоматизації процесів у бізнесі та промисловості?
Програмування дозволяє створювати системи для автоматизації рутинних бізнес-процесів, таких як управління запасами, бухгалтерський облік, обробка замовлень і аналітика. У промисловості програмне забезпечення керує автоматизованими виробничими лініями, контролює якість продукції, оптимізує логістичні процеси та забезпечує віддалене керування обладнанням. Завдяки програмуванню підприємства можуть значно скоротити витрати, підвищити продуктивність і якість роботи.

5. Як програмування впливає на розвиток інформаційних технологій та новітніх технологій?
Програмування є рушійною силою розвитку IT-сфери, оскільки дозволяє створювати нові технології та вдосконалювати існуючі. Воно сприяє розвитку штучного інтелекту, машинного навчання, обробки великих даних, інтернету речей (IoT) та хмарних обчислень. Програмування також відіграє важливу роль у створенні блокчейн-технологій, віртуальної та доповненої реальності, що значно змінює ландшафт сучасних технологій.

6. Що таке мова програмування і які основні типи мов програмування існують?
Мова програмування — це набір синтаксичних та семантичних правил, що визначають, як інструкції для комп'ютера повинні бути написані та виконані. Основні типи мов програмування включають:
- Процедурні мови: Наприклад, C, де програма складається з послідовності інструкцій або процедур.
- Об'єктно-орієнтовані мови: Наприклад, Java, C#, де програми моделюють реальні об'єкти з властивостями та методами.
- Функціональні мови: Наприклад, Haskell, де програми побудовані на основі математичних функцій.
- Логічні мови: Наприклад, Prolog, де програми описують логічні відношення та правила для виведення висновків.

7. Назвіть декілька популярних мов програмування та їх застосування в різних галузях.
- Python: Наукові дослідження, машинне навчання, аналіз даних, веб-розробка.

- JavaScript: Веб-розробка, створення динамічних веб-сторінок та інтерфейсів користувача.
- Java: Розробка корпоративних систем, мобільних додатків, обробка великих даних.
- C++: Вбудовані системи, ігри, системне програмне забезпечення, додатки з високою продуктивністю.
- SQL: Управління базами даних, обробка великих обсягів даних, фінансовий аналіз.

8. Які критерії можна використовувати для класифікації мов програмування?
Мови програмування можна класифікувати за:
- Рівнем абстракції: Високорівневі (Python, Java) і низькорівневі (Assembly).
- Парадигмою програмування: Процедурні, об'єктно-орієнтовані, функціональні, логічні.
- Призначенням: Загального призначення (C++, Python) і спеціалізовані (SQL для роботи з базами даних).
- Синтаксисом: Мови з суворим (C, Java) і вільним синтаксисом (Python).

9. Що таке високорівнева та низькорівнева мови програмування? Наведіть приклади кожного типу.
- Високорівневі мови програмування: Це мови, які надають великий рівень абстракції від апаратного забезпечення, що дозволяє програмістам писати код, який легше розуміти і підтримувати. Приклади: Python, Java, C#.
- Низькорівневі мови програмування: Це мови, які надають менший рівень абстракції та ближчі до машинного коду, що забезпечує більший контроль над апаратним забезпеченням. Приклади: Assembly, C.

10. Як вибрати мову програмування для конкретного проекту? Які критерії слід враховувати при цьому?
Вибір мови програмування для проекту залежить від кількох факторів:
- Цілі проекту: Наприклад, Python підходить для швидкої розробки прототипів, а C++ для додатків з високою продуктивністю.
- Платформа: Java підходить для кросплатформних додатків, а Swift для розробки під iOS.
- Продуктивність: Для завдань, де важлива швидкість і ефективність, краще використовувати C або C++.
- Спільнота та підтримка: Мови з великою спільнотою (Python, JavaScript) пропонують більше бібліотек, фреймворків і документації.

- Масштабованість: Для великих проектів, які потребують модульності та підтримки, обирають мови з хорошою підтримкою об'єктно-орієнтованого програмування (Java, C#).

11. Які основні особливості мови програмування Python?
Python відрізняється такими ключовими особливостями:
- Простота та зручність: Python має простий та зрозумілий синтаксис, що робить його доступним для новачків.
- Інтерпретованість: Python є інтерпретованою мовою, що означає, що код виконується рядок за рядком, без необхідності компіляції.
- Динамічна типізація: Тип змінної визначається автоматично під час виконання програми, що спрощує розробку.
- Велика стандартна бібліотека: Python має багату стандартну бібліотеку, яка включає модулі для роботи з файлами, веб-сервісами, базами даних та іншими завданнями.
- Підтримка кількох парадигм програмування: Python підтримує об'єктно-орієнтоване, процедурне та функціональне програмування.
- Переносимість: Програми на Python можуть виконуватися на різних платформах без зміни коду.
- Активна спільнота: Python має велику спільноту розробників, що забезпечує велику кількість бібліотек, фреймворків та документації.

12. Що таке динамічна типізація у Python, і яке її переваги?
Динамічна типізація означає, що тип змінної визначається автоматично під час виконання програми, а не під час компіляції. Наприклад, у Python можна оголосити змінну без вказання її типу: `x = 10`, де `x` автоматично визначається як ціле число. Це надає такі переваги:
- Гнучкість: Змінні можуть змінювати свій тип в процесі роботи програми, що дозволяє адаптуватися до різних умов.
- Простота: Розробникам не потрібно визначати типи змінних, що спрощує код і зменшує кількість помилок, пов'язаних з типами даних.
- Швидкий розвиток: Можливість швидко створювати прототипи та писати код без необхідності суворої типізації.

13. Які особливості синтаксису Python дозволяють зробити код більш читабельним?
Python приділяє велику увагу читабельності коду, завдяки:
- Індентація: Блоки коду визначаються індентацією (відступами), що змушує розробників писати код, який легко читати та розуміти.
- Простота конструкцій: Мінімум зайвих символів, таких як фігурні дужки чи крапки з комою.

- Зрозумілі імена змінних та функцій: Python заохочує використання зрозумілих імен змінних та функцій, що полегшує розуміння коду.
- Мінімалізм: У Python існує правило "краще одне очевидне рішення, ніж кілька різних", що сприяє однозначності коду.

14. Як Python керує пам'яттю та автоматично вирішує проблеми зі сміттям?
Python використовує механізм автоматичного збору сміття (Garbage Collection) для управління пам'яттю. Основні особливості:
- Лічильник посилань: Кожен об'єкт у Python має лічильник посилань, який збільшується, коли до об'єкта додається нове посилання, і зменшується, коли посилання видаляється. Коли лічильник досягає нуля, об'єкт автоматично видаляється.
- Збір сміття: Окрім лічильника посилань, Python має механізм збору сміття для видалення циклічних посилань, які лічильник не може відслідкувати.
- Керування пам'яттю: Пам'ять виділяється та звільняється автоматично, без необхідності ручного керування, що зменшує кількість помилок, пов'язаних з пам'яттю.

15. Що таке "Pythonic" код, і чому вважається важливим дотримуватися Pythonic стилю програмування?
"Pythonic" код — це код, написаний у стилі, який відповідає принципам та ідеалам мови Python, а саме:
- Простота та зрозумілість: Код має бути простим для читання та зрозумілим для інших розробників.
- Використання вбудованих можливостей: Використання стандартних бібліотек та ідіоматичних виразів, що дозволяють досягти результату ефективніше.
- Ясність та явність: Краще використовувати явний код (який чітко показує, що він робить), ніж приховані або надто складні конструкції.
Дотримання Pythonic стилю допомагає створювати код, який легше підтримувати, розширювати та використовувати повторно.

16. Які основні кроки потрібно виконати для встановлення Python на ваш комп'ютер?
Для встановлення Python потрібно виконати наступні кроки:
1. Завантаження інсталятора: Відвідайте офіційний сайт Python (https://www.python.org) та завантажте останню версію Python для вашої операційної системи.
2. Запуск інсталятора: Запустіть завантажений інсталятор. На Windows важливо відзначити опцію "Add Python to PATH".

3. Налаштування параметрів: Під час інсталяції можна вибрати опції, такі як встановлення `pip` (менеджер пакетів), створення ярликів тощо.
4. Перевірка встановлення: Після завершення інсталяції відкрийте командний рядок (CMD або Terminal) та введіть `python --version` для перевірки успішного встановлення.

17. Які альтернативні методи встановлення Python існують для різних операційних систем?
- Windows: Окрім стандартного інсталятора, можна використовувати менеджери пакетів, такі як Chocolatey або Scoop, для встановлення Python.
- macOS: Python можна встановити через Homebrew (`brew install python`) або використовувати офіційний інсталятор з сайту Python.
- Linux: Python зазвичай встановлений за замовчуванням. Якщо потрібно оновити або встановити додаткову версію, можна використовувати менеджери пакетів, такі як `apt` (Debian/Ubuntu) або `yum` (Fedora/CentOS), або завантажити вихідний код та зібрати Python вручну.

18. Що таке віртуальне середовище Python і чому воно важливе для розробки програм на Python?
Віртуальне середовище Python — це ізольоване середовище, в якому можна встановлювати залежності та пакети для конкретного проекту, не впливаючи на глобальну систему Python або інші проекти. Воно важливе тому, що:
- Запобігання конфліктам: Кожен проект може мати свої версії бібліотек, і віртуальне середовище дозволяє уникати конфліктів між ними.
- Легка керованість: Можливість швидкого видалення або оновлення залежностей без ризику порушення інших проектів.
- Простота розгортання: Легко переносити проект разом з усіма залежностями на інші машини або сервери.

19. Як створити віртуальне середовище Python і активувати його на вашому комп'ютері?
Для створення та активації віртуального середовища виконайте такі кроки:
1. Створення віртуального середовища: У командному рядку виконайте команду `python -m venv myenv`, де `myenv` — ім'я вашого віртуального середовища.
2. Активація середовища:
   - На Windows: `myenv\Scripts\activate`

- На macOS/Linux: `source myenv/bin/activate`
3. Встановлення пакетів: Після активації можна використовувати `pip` для встановлення необхідних пакетів у цьому середовищі.

20. Які є основні інтегровані середовища розробки (IDE) для програмування на Python, і які переваги та недоліки кожного з них?

Інтегровані середовища розробки (IDE) значно полегшують процес написання, налагодження та тестування коду на Python. Ось кілька популярних IDE для Python, а також їхні переваги та недоліки:

1. PyCharm
Переваги:
- Потужні інструменти для розробки: PyCharm пропонує розширені інструменти для рефакторингу, автозаповнення коду, аналізу коду та налагодження.
- Підтримка фреймворків: Підтримка популярних Python-фреймворків, таких як Django, Flask, і Pyramid.
- Інтеграція з системами контролю версій: PyCharm має вбудовану підтримку для Git, Mercurial та інших систем контролю версій.
- Розширюваність: Можливість встановлення додаткових плагінів для розширення функціональності.

Недоліки:
- Ресурсоємність: PyCharm є досить важким і може споживати багато оперативної пам'яті та ресурсів CPU, що може бути проблемою для слабких комп'ютерів.
- Ціна: Повна версія PyCharm (Professional Edition) є платною. Хоча існує безкоштовна версія (Community Edition), вона має обмежені можливості.

2. Visual Studio Code (VS Code)
Переваги:
- Легка вага та швидкодія: VS Code є швидким і легким, що робить його придатним для використання навіть на старих машинах.
- Розширюваність: Велика кількість плагінів, включаючи підтримку Python через офіційне розширення Python від Microsoft.
- Вбудований термінал: Зручний вбудований термінал, який дозволяє працювати з командним рядком безпосередньо в IDE.
- Інтеграція з системами контролю версій: Вбудована підтримка для Git, а також можливість інтеграції з іншими системами контролю версій через плагіни.

Недоліки:
- Обмежені можливості для великих проектів: Хоча VS Code відмінно підходить для невеликих та середніх проектів, для великих та складних проектів він може бути менш ефективним у порівнянні з більш спеціалізованими IDE, такими як PyCharm.
- Не завжди інтуїтивний інтерфейс: Потребує певного налаштування та встановлення плагінів для максимального використання його можливостей, що може бути незручним для новачків.

3. Jupyter Notebook
Переваги:
- Інтерактивність: Відмінно підходить для наукових досліджень, аналізу даних та навчання, оскільки дозволяє виконувати код блоками і миттєво бачити результат.
- Підтримка Markdown: Можливість додавати текстові коментарі, формули та графіки, що робить його ідеальним для створення інтерактивних презентацій.
- Популярність в науковому середовищі: Широко використовується серед дослідників, аналітиків даних та вчителів для наукових обчислень та демонстрацій.

Недоліки:
- Обмежений функціонал для розробки великих проектів: Jupyter Notebook не є повноцінним IDE, що робить його менш придатним для розробки великих програмних проектів.
- Відсутність підтримки для складного рефакторингу: Не підтримує складні можливості для рефакторингу та інтеграції, які пропонують традиційні IDE.

4. Spyder
Переваги:
- Оптимізований для наукових обчислень: Spyder інтегрований з бібліотеками для наукових обчислень (такими як NumPy, SciPy, Matplotlib) і має інструменти для роботи з масивами даних.
- Вбудовані консолі: Включає інтегровану IPython-консоль для виконання інтерактивного коду.
- Легкий у використанні: Має простий інтерфейс, який зручний для новачків і досвідчених розробників.

Недоліки:

- Обмежена розширюваність: Хоча Spyder добре працює з науковими обчисленнями, він менш гнучкий для інших типів проектів і не підтримує плагіни на такому рівні, як інші IDE.
- Відносно повільний: Може бути повільнішим у порівнянні з іншими легковаговими редакторами, особливо при роботі з великими даними.

5. Thonny
Переваги:
- Простота для початківців: Thonny спеціально створений для навчання програмуванню на Python, з простим інтерфейсом та вбудованим налагоджувачем.
- Автоматичне управління середовищами: Thonny автоматично створює віртуальне середовище для кожного проекту.
- Просте встановлення та використання: Мінімум налаштувань і швидкий старт.

Недоліки:
- Обмежений функціонал: Хоча Thonny відмінно підходить для новачків, він не надає розширених можливостей для роботи над великими та складними проектами.
- Відсутність розширюваності: Thonny має дуже обмежену кількість плагінів та розширень.

6. IDLE (Integrated Development and Learning Environment)
Переваги:
- Простота: Вбудоване в дистрибутив Python середовище, що не потребує додаткової інсталяції.
- Ідеально для навчання: Підходить для вивчення основ програмування та написання простих скриптів.

Недоліки:
- Обмежені можливості: IDLE надає лише базовий функціонал для написання та налагодження коду, що робить його непридатним для складних проектів.
- Застарілий інтерфейс: Інтерфейс IDLE виглядає старомодно і менш зручний у порівнянні з сучасними IDE.

Висновок:
Вибір IDE для програмування на Python залежить від потреб розробника та специфіки проекту. Для великих професійних проектів найкраще підходить PyCharm, для роботи з даними та наукових досліджень — Jupyter Notebook або Spyder, а для навчання і простих

проектів — Thonny або IDLE. Visual Studio Code є універсальним варіантом, який може підійти для багатьох типів проектів завдяки своїй розширюваності та гнучкості.

Лекція №2. Основи Python: типи даних та змінні

2.1 Знайомство з основними типами даних у Python (числа, рядки, списки, кортежі, словники)

Python є популярною мовою програмування, яка має різноманітні вбудовані типи даних. У цій темі ми розглянемо основні типи даних, зокрема числа, рядки, списки, кортежі та словники. Кожен з них має свої особливості та використовується для зберігання різного типу інформації.

Числа

У Python є кілька типів числових даних:

1. int – цілі числа. Вони можуть бути як позитивними, так і негативними, а також мати довільну довжину.

```
a = 5
b = -10
c = 123456789
```

2. float – числа з плаваючою крапкою (десяткові числа). Використовуються для представлення дійсних чисел.

```
x = 3.14
y = -0.001
```

3. complex – комплексні числа, які мають реальну та уявну частини.

```
z = 2 + 3j
```

Рядки (str)

Рядки у Python використовуються для зберігання текстових даних. Вони оточені одинарними (' ') або подвійними (" ") лапками.

```
name = "John"
greeting = 'Hello, World!'
```

Рядки є незмінними, тобто після створення їх неможливо змінити.

Списки (list)

Списки є впорядкованими змінними колекціями елементів. Вони можуть містити елементи різних типів даних.

```
numbers = [1, 2, 3, 4, 5]
mixed = [1, "two", 3.0, [4, 5]]
```

Списки підтримують індексацію, нарізання, зміну та додавання елементів.

Кортежі (tuple)

Кортежі схожі на списки, але вони є незмінними, тобто після створення їх не можна змінити. Використовуються для зберігання групи елементів.

```
point = (1, 2)
colors = ("red", "green", "blue")
```

Словники (dict)

Словники є невпорядкованими колекціями пар "ключ-значення". Ключі повинні бути унікальними та незмінними (наприклад, рядки, числа, кортежі).

```
student = {
    "name": "Alice",
    "age": 21,
    "grades": [88, 92, 79]
}
```

Доступ до значень здійснюється через ключі:

```
name = student["name"]
age = student["age"]
```

Знайомство з основними типами даних у Python є важливим кроком у вивченні цієї мови програмування. Числа, рядки, списки, кортежі та словники використовуються для зберігання та маніпуляції даними

різного типу, і розуміння їх особливостей допомагає ефективніше писати та читати код на Python.

2.2 Поняття змінних та їх призначення

Що таке змінна?

Змінна – це іменована область пам'яті, яка використовується для зберігання даних. У програмуванні змінні дозволяють зберігати значення, які можна використовувати та змінювати протягом виконання програми. Змінні діють як контейнери для даних, що дозволяє писати більш гнучкий та зрозумілий код.

Призначення змінних

Основне призначення змінних – це можливість зберігати дані для подальшого використання та обробки. Вони дозволяють:

1. Зберігати інформацію: Ви можете зберігати різні типи даних (числа, рядки, списки, тощо) і використовувати їх у подальших обчисленнях або логічних операціях.

```
name = "Alice"
age = 25
```

2. Змінювати значення: Змінні дозволяють змінювати збережене значення без потреби переписувати весь код.

```
counter = 0
counter = counter + 1  # тепер counter дорівнює 1
```

3. Підвищувати читабельність коду: Використання зрозумілих імен змінних робить код легшим для розуміння та підтримки.

```
height_in_meters = 1.75
weight_in_kg = 68
bmi = weight_in_kg / (height_in_meters  2)
```

4. Управляти даними: Ви можете організовувати і структурувати дані за допомогою змінних, що допомагає краще контролювати потік даних у програмі.

Оголошення змінних у Python

У Python змінні оголошуються просто шляхом присвоєння їм значень за допомогою оператора `=`:

```
x = 10
y = "Hello, World!"
is_valid = True
```

Іменування змінних

Імена змінних повинні бути описовими та зрозумілими. Дотримуйтесь таких правил:

- Імена змінних можуть містити літери, цифри та символи підкреслення (_).
- Ім'я змінної повинно починатися з літери або підкреслення, але не з цифри.
- Імена змінних чутливі до регістру, тобто `Variable` і `variable` – це різні змінні.
- Використовуйте змістовні імена для змінних, щоб код був легшим для розуміння.

Приклади використання змінних

```
# Приклад 1: Збереження даних
name = "Bob"
age = 30

# Приклад 2: Використання змінних у розрахунках
```

```
length = 5
width = 3
area = length * width  # обчислення площі

# Приклад 3: Зміна значення змінної
score = 10
score = score + 5  # тепер score дорівнює 15
```

Змінні є фундаментальною частиною програмування у Python. Вони забезпечують спосіб зберігання і маніпуляції даними, що дозволяє створювати динамічні та гнучкі програми. Правильне використання змінних робить код більш зрозумілим і зручним для підтримки, що є важливим аспектом ефективного програмування.

2.3 Операції з різними типами даних

У Python різні типи даних підтримують різноманітні операції. Давайте розглянемо основні операції, які можна виконувати з числами, рядками, списками, кортежами та словниками.

Числа

Числа у Python включають цілі числа (int), числа з плаваючою крапкою (float) і комплексні числа (complex). Ось основні арифметичні операції, які можна виконувати з числами:

1. Додавання

```
x = 5
y = 3
result = x + y  # result дорівнює 8
```

2. Віднімання

```
result = x – y  # result дорівнює 2
```

3. Множення

```
result = x * y  # result дорівнює 15
```

4. Ділення

```
result = x / y  # result дорівнює 1.666...
```

5. Цілочисельне ділення

```
result = x // y  # result дорівнює 1
```

6. Залишок від ділення

```
result = x % y  # result дорівнює 2
```

7. Піднесення до степеня

```
result = x ** y  # result дорівнює 125
```

Рядки

Рядки (str) підтримують безліч операцій для маніпуляції текстовими даними:

1. Конкатенація (об'єднання)

```
hello = "Hello"
world = "World"
```

```
greeting = hello + " " + world  # greeting дорівнює "Hello World"
```

2. Повторення

```
laugh = "ha"
multiple_laugh = laugh * 3  # multiple_laugh дорівнює "hahaha"
```

3. Індексація

```
char = greeting[1]  # char дорівнює "e"
```

4. Нарізання (slicing)

```
substring = greeting[0:5]  # substring дорівнює "Hello"
```

5. Методи рядків

```
upper_greeting = greeting.upper()  # upper_greeting дорівнює "HELLO WORLD"
```

Списки

Списки (list) підтримують різні операції для роботи з колекціями елементів:

1. Додавання елемента

```
numbers = [1, 2, 3]
numbers.append(4)  # numbers тепер [1, 2, 3, 4]
```

2. Видалення елемента

```
numbers.remove(2)  # numbers тепер [1, 3, 4]
```

3. Індексація та нарізання

```
first_element = numbers[0]  # first_element дорівнює 1
sublist = numbers[1:3]  # sublist дорівнює [3, 4]
```

4. Додавання списків

```
more_numbers = [5, 6]
all_numbers = numbers + more_numbers  # all_numbers дорівнює [1, 3, 4, 5, 6]
```

Кортежі

Кортежі (tuple) є незмінними, але підтримують багато операцій, аналогічних спискам:

1. Індексація та нарізання

```
colors = ("red", "green", "blue")
first_color = colors[0]  # first_color дорівнює "red"
some_colors = colors[1:3]  # some_colors дорівнює ("green", "blue")
```

2. Конкатенація кортежів

```
more_colors = ("yellow", "purple")
all_colors = colors + more_colors  # all_colors дорівнює ("red", "green", "blue", "yellow", "purple")
```

Словники

Словники (dict) дозволяють зберігати дані у форматі ключ-значення і підтримують такі операції:

1. Додавання пари ключ-значення

```
student = {"name": "Alice", "age": 21}
student["grade"] = "A"  # student тепер {"name": "Alice", "age": 21, "grade": "A"}
```

2. Видалення пари ключ-значення

```
del student["age"]  # student тепер {"name": "Alice", "grade": "A"}
```

3. Отримання значення за ключем

```
name = student["name"]  # name дорівнює "Alice"
```

4. Методи словників

```
keys = student.keys()  # keys дорівнює dict_keys(['name', 'grade'])
values = student.values()  # values дорівнює dict_values(['Alice', 'A'])
```
`

Розуміння операцій з різними типами даних у Python є ключовим аспектом ефективного програмування. Використання відповідних операцій для маніпуляції числами, рядками, списками, кортежами та словниками дозволяє створювати потужні та гнучкі програми, що легко читаються та підтримуються.

Питання для перевірки знань по темі лекції

1. Числа

– Питання: Які три основні типи числових даних підтримує Python? Наведіть приклади кожного з них.

– Відповідь: Python підтримує цілі числа (int), числа з плаваючою крапкою (float) і комплексні числа (complex).

```
int_example = 42
float_example = 3.14
complex_example = 1 + 2j
```

2. Рядки

– Питання: Як створити рядок у Python? Чим відрізняється рядок, оголошений в одинарних лапках, від рядка, оголошеного в подвійних лапках?

– Відповідь: Рядок створюється за допомогою одинарних (' ') або подвійних (" ") лапок. Різниці між ними немає, обидва способи є еквівалентними.

```
single_quote_str = 'Hello'
double_quote_str = "World"
```

3. Списки

– Питання: Як створити список у Python? Як додати новий елемент до списку?

– Відповідь: Список створюється за допомогою квадратних дужок. Новий елемент додається за допомогою методу `.append()`.

```
my_list = [1, 2, 3]
my_list.append(4)  # тепер my_list дорівнює [1, 2, 3, 4]
```

4. Кортежі

– Питання: Що таке кортеж і як його створити? Чим кортежі відрізняються від списків?

– Відповідь: Кортежі створюються за допомогою круглих дужок і є незмінними, тобто їх не можна змінювати після створення.

```
my_tuple = (1, 2, 3)
```

```
# Спроба змінити кортеж викличе помилку
# my_tuple[0] = 0  # Помилка
```

5. Словники

– Питання: Як створити словник у Python? Як отримати значення за ключем?

– Відповідь: Словник створюється за допомогою фігурних дужок і зберігає дані у форматі ключ-значення. Значення отримується за ключем.

```
my_dict = {"name": "Alice", "age": 25}
age = my_dict["age"]  # age дорівнює 25
```

6. Числові операції

– Питання: Яка різниця між операціями ділення `/` та цілочисельного ділення `//` у Python?

– Відповідь: Операція `/` повертає результат ділення з плаваючою крапкою, а `//` повертає лише цілу частину від ділення.

```
result1 = 7 / 2  # result1 дорівнює 3.5
result2 = 7 // 2  # result2 дорівнює 3
```

7. Рядкові операції

– Питання: Як можна об'єднати два рядки в Python? Як повторити рядок кілька разів?

– Відповідь: Рядки об'єднуються за допомогою операції `+`. Повторити рядок можна за допомогою операції `*`.

```
str1 = "Hello"
str2 = "World"
combined_str = str1 + " " + str2  # combined_str дорівнює "Hello World"
```

```
repeated_str = str1 * 3  # repeated_str дорівнює "HelloHelloHello"
```

8. Операції зі списками

– Питання: Як можна об'єднати два списки в один? Як отримати підсписок із списку?

– Відповідь: Списки об'єднуються за допомогою операції `+`. Підсписок можна отримати за допомогою нарізання (slicing).

```
list1 = [1, 2, 3]
list2 = [4, 5]
combined_list = list1 + list2  # combined_list дорівнює [1, 2, 3, 4, 5]
sublist = combined_list[1:4]  # sublist дорівнює [2, 3, 4]
```

9. Індексація

– Питання: Як отримати перший і останній елемент списку або кортежу?

– Відповідь: Перший елемент отримується за допомогою індексу `0`, останній елемент – за допомогою індексу `-1`.

```
my_list = [1, 2, 3, 4]
first_element = my_list[0]  # first_element дорівнює 1
last_element = my_list[-1]  # last_element дорівнює 4
```

10. Методи словників

– Питання: Як отримати список усіх ключів і список усіх значень у словнику?

– Відповідь: Використовуються методи `.keys()` та `.values()`.

```
my_dict = {"name": "Alice", "age": 25, "grade": "A"}
keys = my_dict.keys()  # keys дорівнює dict_keys(['name', 'age', 'grade'])
```

values = my_dict.values() # values дорівнює dict_values(['Alice', 25, 'A'])

11. Оголошення змінних

– Питання: Як оголосити змінну у Python та присвоїти їй значення?

– Відповідь: Змінна оголошується шляхом присвоєння значення за допомогою оператора `=`. Наприклад:

```
x = 10
name = "Alice"
```

12. Типи змінних

– Питання: Які типи даних можуть зберігатися у змінних Python? Наведіть приклади.

– Відповідь: У змінних Python можуть зберігатися різні типи даних, такі як int, float, str, list, tuple, dict тощо.

```
age = 25  # int
pi = 3.14  # float
greeting = "Hello"  # str
```

13. Іменування змінних

– Питання: Які правила слід дотримувати при іменуванні змінних у Python?

– Відповідь: Імена змінних можуть містити літери, цифри та символи підкреслення (_), повинні починатися з літери або підкреслення, і чутливі до регістру.

```
my_variable = 10
_hidden_variable = 20
```

14. Присвоєння значень

– Питання: Що станеться, якщо присвоїти нове значення вже існуючій змінній?

– Відповідь: Змінна набуде нового значення, замінюючи попереднє.

```
x = 5
x = 10  # тепер x дорівнює 10
```

15. Використання змінних у виразах

– Питання: Як використовуються змінні у математичних виразах? Наведіть приклад.

– Відповідь: Змінні можуть використовуватися у математичних виразах для обчислень.

```
a = 5
b = 3
result = a + b  # result дорівнює 8
```

16. Типи даних змінних

– Питання: Як можна визначити тип даних змінної у Python?

– Відповідь: Використовується функція `type()`.

```
x = 42
print(type(x))  # <class 'int'>
```

17. Зміна типу змінної

– Питання: Що відбувається, коли змінній присвоюється значення іншого типу?

– Відповідь: Змінна набуває нового типу, відповідно до присвоєного значення.

```
var = 10  # int
var = "Hello"  # тепер var має тип str
```

18. Глобальні та локальні змінні

– Питання: Яка різниця між глобальними та локальними змінними?

– Відповідь: Глобальні змінні оголошуються поза функціями і доступні у всій програмі, тоді як локальні змінні оголошуються всередині функцій і доступні лише в межах цієї функції.

```
global_var = "I am global"

def my_function():
    local_var = "I am local"
    print(global_var)  # доступна всередині функції
    print(local_var)  # доступна всередині функції

my_function()
print(global_var)  # доступна поза функцією
# print(local_var)  # викличе помилку
```

19. Змінні та константи

– Питання: Чи є у Python константи, і як зазвичай позначаються такі змінні?

– Відповідь: У Python немає вбудованого поняття констант, але змінні, які не повинні змінюватися, зазвичай позначаються великими літерами.

```
PI = 3.14159
MAX_USERS = 100
```

20. Перетворення типів даних

– Питання: Як можна перетворити тип даних змінної у Python? Наведіть приклад.

– Відповідь: Перетворення типів даних здійснюється за допомогою відповідних функцій, таких як `int()`, `float()`, `str()`.

```
python
num_str = "123"
num_int = int(num_str)  # тепер num_int дорівнює 123 (тип int)
```

21. Числові операції

– Питання: Яка різниця між операціями `**` і `pow()` у Python?

– Відповідь: Обидві операції виконують піднесення до степеня, але `**` – це оператор, а `pow()` – вбудована функція. Крім того, `pow()` може приймати три аргументи для обчислення степеня з модулем.

```
result1 = 2 ** 3  # result1 дорівнює 8
result2 = pow(2, 3)  # result2 дорівнює 8
result3 = pow(2, 3, 3)  # result3 дорівнює 2 (8 % 3)
```

22. Рядкові операції

– Питання: Як об'єднати список рядків в один рядок з певним роздільником?

– Відповідь: Використовується метод `join()`.

```
words = ["Hello", "world"]
sentence = " ".join(words)  # sentence дорівнює "Hello world"
```

23. Операції зі списками

– Питання: Як додати кілька елементів до списку за один виклик?

– Відповідь: Використовується метод `extend()`.

```
my_list = [1, 2, 3]
my_list.extend([4, 5, 6])  # my_list тепер [1, 2, 3, 4, 5, 6]
```

24. Операції з кортежами

– Питання: Чи можна змінити елемент у кортежі? Якщо ні, то як можна обійти це обмеження?

– Відповідь: Кортежі є незмінними, тому елемент у них змінити не можна. Однак можна створити новий кортеж на основі старого з внесеними змінами.

```
my_tuple = (1, 2, 3)
```

```
new_tuple = my_tuple[:1] + (4,) + my_tuple[2:]  # new_tuple дорівнює (1, 4, 3)
```

25. Операції зі словниками

– Питання: Як можна отримати всі ключі та значення зі словника?

– Відповідь: Використовуються методи `keys()` та `values()`.

```
my_dict = {"name": "Alice", "age": 25}
keys = my_dict.keys()  # keys дорівнює dict_keys(['name', 'age'])
values = my_dict.values()  # values дорівнює dict_values(['Alice', 25])
```

26. Змішування типів даних

– Питання: Що станеться, якщо спробувати об'єднати рядок і число за допомогою операції `+`?

– Відповідь: Виникне помилка типу, оскільки Python не дозволяє об'єднувати рядки та числа напряму. Потрібно перетворити число на рядок.

```
result = "The answer is " + str(42)  # result дорівнює "The answer is 42"
```

27. Індексація та нарізання

– Питання: Як отримати останні три елементи списку за допомогою нарізання?

– Відповідь: Використовується негативний індекс та синтаксис нарізання.

```
my_list = [1, 2, 3, 4, 5]
last_three = my_list[-3:]  # last_three дорівнює [3, 4, 5]
```

28. Перетворення типів даних

– Питання: Як перетворити список кортежів у словник?

– Відповідь: Використовується конструктор `dict()`.

```
list_of_tuples = [("name", "Alice"), ("age", 25)]
my_dict = dict(list_of_tuples)  # my_dict дорівнює {'name': 'Alice', 'age': 25}
```

29. Методи рядків
– Питання: Як перевірити, чи рядок починається з певного підрядка?
– Відповідь: Використовується метод `startswith()`.

```
greeting = "Hello, world!"
result = greeting.startswith("Hello")  # result дорівнює True
```

20. Методи списків
– Питання: Як видалити елемент із списку за його значенням?
– Відповідь: Використовується метод `remove()`.

```
my_list = [1, 2, 3, 4, 5]
my_list.remove(3)  # my_list тепер [1, 2, 4, 5]
```

Лекція №3. Умовні оператори та цикли

3.1 Використання умовних операторів if, elif, else.

Умовні оператори в Python дозволяють програмі приймати рішення та виконувати різні дії залежно від виконання певних умов. Це важливий інструмент для створення логічних структур в коді.

Основні концепції

1. **Оператор `if`**:
– Використовується для перевірки, чи виконується певна умова. Якщо умова істинна (True), виконується блок коду під `if`.

```
if умова:
    # код виконується, якщо умова істинна
```

2. **Оператор `elif`**:
– Скорочення від "else if". Використовується для перевірки додаткових умов, якщо попередня умова `if` або `elif` не була істинною.

```
if умова1:
    # код виконується, якщо умова1 істинна
elif умова2:
    # код виконується, якщо умова1 не істинна, а умова2 істинна
```

3. **Оператор `else`**:
– Використовується для виконання блоку коду, якщо жодна з попередніх умов не була істинною.

```
if умова1:
    # код виконується, якщо умова1 істинна
elif умова2:
    # код виконується, якщо умова1 не істинна, а умова2 істинна
else:
    # код виконується, якщо жодна з умов не істинна
```

Приклади використання

1. **Простий приклад з `if`**:

```
age = 20
if age >= 18:
    print("Ви повнолітній.")
```

У цьому прикладі перевіряється, чи вік користувача більший або дорівнює 18. Якщо умова істинна, виводиться повідомлення "Ви повнолітній."

2. **Приклад з `if` та `else`**:

```
age = 16
if age >= 18:
    print("Ви повнолітній.")
else:
    print("Ви неповнолітній.")
```

Тут перевіряється та ж умова, але якщо вона не істинна, виконується блок коду під `else`.

3. **Приклад з `if`, `elif` та `else`**:

```
score = 85
if score >= 90:
    print("Відмінно!")
elif score >= 75:
    print("Добре!")
elif score >= 50:
    print("Задовільно.")
else:
    print("Незадовільно.")
```

У цьому прикладі використовуються кілька умов для визначення оцінки на основі набраних балів. Програма перевіряє умови послідовно, і виконується перший блок коду з істинною умовою.

Умовні оператори `if`, `elif`, `else` є важливими складовими Python, що дозволяють створювати логічні структури в коді. Вони надають можливість програмі приймати рішення на основі заданих умов та виконувати відповідні дії. Це основа для побудови більш складних алгоритмів та програм.

3.2 Робота з циклами: for та while.

Цикли дозволяють виконувати один і той самий блок коду багаторазово, що є важливим інструментом для автоматизації повторюваних завдань. Python підтримує два основних типи циклів: `for` та `while`.

Цикл `for`

Цикл `for` використовується для ітерації через послідовності (такі як списки, кортежі, рядки) або інші ітерабельні об'єкти. Це дозволяє виконати блок коду для кожного елементу послідовності.

Синтаксис:

```
for елемент in послідовність:
    # блок коду, який буде виконуватись для кожного елементу
```

Приклади:

1. **Ітерація через список:**

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

У цьому прикладі цикл `for` ітерує через кожен елемент списку `fruits` і виводить його значення.

2. **Ітерація через рядок:**

```
for letter in "Python":
    print(letter)
```

Цей приклад виводить кожну букву в рядку "Python".

3. **Використання функції `range`:**

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

Функція `range(5)` створює послідовність чисел від 0 до 4, і цикл `for` ітерує через цю послідовність, виводячи кожне число.

Цикл `while`

Цикл `while` виконує блок коду, поки умова залишається істинною. Він продовжує ітерацію доти, доки умова не стане хибною (False).

Синтаксис:

```
while умова:
    # блок коду, який буде виконуватись поки умова істинна
```

Приклади:

1. **Простий приклад з лічильником:**

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

У цьому прикладі змінна `i` збільшується на 1 при кожній ітерації циклу, доки не досягне значення 5.

2. **Безкінечний цикл (будьте обережні!):**

```
while True:
    print("Цей цикл буде виконуватись вічно")
```

Цей приклад показує безкінечний цикл, оскільки умова завжди істинна. Такий цикл можна зупинити тільки вручну (наприклад, натиснувши Ctrl+C).

Вкладені цикли

Цикли можуть бути вкладеними один в один, що дозволяє виконувати більш складні ітерації.

Приклад:

```
for i in range(3):
    for j in range(2):
        print(f"i={i}, j={j}")
```

У цьому прикладі зовнішній цикл `for` ітерує три рази, а внутрішній цикл `for` – два рази при кожній ітерації зовнішнього циклу, виводячи значення `i` та `j`.

Цикли `for` та `while` є важливими інструментами в Python, що дозволяють автоматизувати повторювані завдання. Цикл `for` зручний для ітерації через послідовності, тоді як цикл `while` корисний, коли кількість ітерацій заздалегідь невідома і залежить від виконання певної умови. Розуміння цих конструкцій є ключовим для ефективного програмування на Python.

3.3 Використання операторів вибору та циклічних конструкцій для вирішення завдань.

Оператори вибору (умовні оператори) та циклічні конструкції (цикли) є основними елементами для створення логічних і повторюваних структур в програмах. Вони дозволяють приймати рішення на основі

умов і виконувати код багаторазово, що є критично важливим для автоматизації різних завдань та побудови складних алгоритмів.

Умовні оператори

Умовні оператори (`if`, `elif`, `else`) дозволяють програмі приймати рішення на основі виконання певних умов. Вони використовуються для керування потоком виконання програми.

Приклад: Визначення парності числа

```
number = int(input("Введіть число: "))
if number % 2 == 0:
    print(f"{number} є парним числом.")
else:
    print(f"{number} є непарним числом.")
```

У цьому прикладі програма перевіряє, чи є введене число парним або непарним, використовуючи оператор `if-else`.

Циклічні конструкції

Цикли (`for`, `while`) дозволяють виконувати один і той самий блок коду багаторазово, що є важливим для повторюваних завдань.

Приклад: Виведення чисел від 1 до 10

```
for i in range(1, 11):
    print(i)
```

У цьому прикладі цикл `for` ітерує через послідовність чисел від 1 до 10 і виводить їх.

Комбінація умовних операторів та циклів

Використання умовних операторів та циклів разом дозволяє створювати складні алгоритми для вирішення різних завдань.

Приклад: Виведення парних чисел від 1 до 20

```
for i in range(1, 21):
    if i % 2 == 0:
        print(i)
```

У цьому прикладі цикл `for` ітерує через числа від 1 до 20, а умовний оператор `if` перевіряє, чи є число парним. Якщо число парне, воно виводиться на екран.

Приклад: Пошук простих чисел у діапазоні

```
start = 2
end = 50

for num in range(start, end + 1):
    is_prime = True
    for i in range(2, num):
        if num % i == 0:
            is_prime = False
            break
    if is_prime:
        print(num)
```

У цьому прикладі програма використовує вкладений цикл для перевірки, чи є число простим. Зовнішній цикл ітерує через заданий

діапазон, а внутрішній цикл перевіряє, чи ділиться число на будь-яке інше число, крім 1 і самого себе.

Приклад: Обробка введення користувача з перевіркою на правильність

```
while True:
    user_input = input("Введіть позитивне число (або 'exit' для виходу): ")
    if user_input.lower() == 'exit':
        break
    elif not user_input.isdigit() or int(user_input) <= 0:
        print("Некоректне введення. Спробуйте ще раз.")
    else:
        print(f"Ви ввели число: {user_input}")
```

У цьому прикладі цикл `while` продовжує запитувати у користувача введення, поки не буде введено команду `exit`. Умовні оператори перевіряють, чи введене значення є позитивним числом і надають відповідний зворотний зв'язок.

Оператори вибору та циклічні конструкції є фундаментальними інструментами для вирішення завдань в Python. Вони дозволяють програмам приймати рішення та виконувати повторювані дії, що є критичним для побудови ефективних і складних алгоритмів. Комбінація цих елементів дозволяє вирішувати широкий спектр завдань, від простих до дуже складних.

Питання для перевірки знань по темі лекції

1. Що таке умовні оператори в Python і для чого вони використовуються?

Відповідь: Умовні оператори дозволяють програмі приймати рішення та виконувати різні дії залежно від виконання певних умов.

2. Який синтаксис оператора `if` в Python?

Відповідь:

```
if умова:
    # код виконується, якщо умова істинна
```

3. Як використовувати оператор `elif` в Python? Наведіть приклад.

Відповідь:

```
if умова1:
    # код виконується, якщо умова1 істинна
elif умова2:
    # код виконується, якщо умова1 не істинна, а умова2 істинна
```

4. Чим відрізняється оператор `else` від `elif`?

Відповідь: Оператор `else` виконується, якщо жодна з попередніх умов `if` або `elif` не була істинною, тоді як `elif` перевіряє додаткову умову.

5. Напишіть приклад коду, який визначає, чи є число додатнім, від'ємним або нулем, використовуючи `if`, `elif`, `else`.

Відповідь:

```
number = int(input("Введіть число: "))
if number > 0:
    print("Число додатнє.")
elif number < 0:
    print("Число від'ємне.")
else:
    print("Число дорівнює нулю.")
```

6. Що буде виведено на екран після виконання наступного коду?

```
x = 10
```

```
if x > 5:
    print("Більше 5")
elif x > 8:
    print("Більше 8")
else:
    print("Менше або дорівнює 5")
```

Відповідь: Виведено буде "Більше 5", оскільки перша умова `if x > 5` істинна, а далі перевірка не виконується.

7. Як можна спростити наступний код?

```
if x > 10:
    print("Більше 10")
elif x >= 10:
    print("Дорівнює 10")
else:
    print("Менше 10")
```

Відповідь:

```
if x > 10:
    print("Більше 10")
elif x == 10:
    print("Дорівнює 10")
else:
    print("Менше 10")
```

8. Що таке вкладені умовні оператори і як їх використовувати? Наведіть приклад.

Відповідь: Вкладені умовні оператори – це умовні оператори всередині інших умовних операторів. Вони використовуються для більш складної логіки.

```
x = 10
if x > 5:
    if x < 15:
        print("x між 5 і 15")
    else:
        print("x більше або дорівнює 15")
else:
    print("x менше або дорівнює 5")
```

9. Що означає ключове слово `elif` в Python?

Відповідь: `elif` – це скорочення від "else if". Воно використовується для перевірки додаткової умови, якщо попередня умова `if` або `elif` не була істинною.

10. Чи можна використовувати `if`, `elif` та `else` безпосередньо в одній конструкції? Якщо так, наведіть приклад.

Відповідь: Так, можна.

```
temperature = 25
if temperature > 30:
    print("Дуже жарко")
elif temperature > 20:
    print("Тепло")
elif temperature > 10:
    print("Прохолодно")
else:
    print("Холодно")
```

11. Який синтаксис циклу `for` в Python?

Відповідь:

```
for елемент in послідовність:
    # блок коду, який буде виконуватись для кожного елементу
```

12. Який синтаксис циклу `while` в Python?

Відповідь:

```
while умова:
    # блок коду, який буде виконуватись поки умова істинна
```

13. Що виведе наступний код?

```
for i in range(3):
    print(i)
```

Відповідь: Виведе:

```
0
1
2
```

14. Як вийти з циклу `while` достроково?

Відповідь: Використати оператор `break`.

15. Що робить оператор `continue` в циклах `for` та `while`?

Відповідь: Пропускає поточну ітерацію циклу та переходить до наступної.

16. Що виведе наступний код?

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

Відповідь: Виведе:

```
1
2
4
```

5

17. Як створити цикл `for`, який ітерує через список [10, 20, 30, 40] і виводить кожен елемент?

Відповідь:

```
my_list = [10, 20, 30, 40]
for item in my_list:
    print(item)
```

18. Що таке безкінечний цикл і як його уникнути?

Відповідь: Безкінечний цикл виконується без кінця через умову, яка завжди істинна. Щоб його уникнути, необхідно забезпечити зміну умови всередині циклу або використовувати оператор `break`.

19. Напишіть цикл `while`, який буде зупинятись після введення користувачем слова "stop".

Відповідь:

```
while True:
    user_input = input("Введіть щось (або 'stop' для виходу): ")
    if user_input.lower() == 'stop':
        break
```

20. Що виведе наступний код?

```
for i in range(2, 10, 2):
    print(i)
```

Відповідь: Виведе:

2
4
6
8

21. Що таке оператор вибору в Python і як він працює?

Відповідь:

Оператор вибору дозволяє програмі приймати рішення та виконувати різні дії залежно від виконання певних умов. Основні оператори вибору: `if`, `elif`, `else`.

22. Який синтаксис поєднання умовного оператора `if` і циклу `for`?
Відповідь:

```
for елемент in послідовність:
    if умова:
        # блок коду, який виконується, якщо умова істинна
```

23. Як використовувати цикл `while` з умовним оператором `if` для перевірки введення користувача на позитивне число?
Відповідь:

```
while True:
    number = int(input("Введіть позитивне число: "))
    if number > 0:
        break
    else:
        print("Число повинно бути позитивним. Спробуйте ще раз.")
```

24. Напишіть код, який використовує `for`, `if`, і `else` для виведення парних чисел від 1 до 20.
Відповідь:

```
for i in range(1, 21):
    if i % 2 == 0:
        print(i)
```

25. Що виведе наступний код?

```
for i in range(5):
    if i == 3:
        print("Знайдено 3")
```

```
    else:
        print(i)
```
Відповідь:
```
0
1
2
Знайдено 3
4
```

26. Як вийти з циклу `for`, якщо знайдено певне значення, використовуючи умовний оператор?

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

27. Напишіть програму, яка використовує цикл `while` і умовні оператори для вгадування числа, поки користувач не введе правильне значення (наприклад, 7).

Відповідь:
```
secret_number = 7
while True:
    guess = int(input("Вгадайте число: "))
    if guess == secret_number:
        print("Ви вгадали!")
        break
    else:
        print("Спробуйте ще раз.")
```

28. Що таке вкладені умовні оператори і як їх використовувати в циклах? Наведіть приклад.

Відповідь: Вкладені умовні оператори – це оператори `if` всередині інших операторів `if`, які дозволяють створювати більш складні логічні структури.

```
for i in range(5):
    if i % 2 == 0:
        if i == 0:
            print("Нуль")
        else:
            print("Парне число:", i)
    else:
        print("Непарне число:", i)
```

29. Як обробити виняткові ситуації (помилки) під час введення користувача, використовуючи цикл `while` і умовні оператори?

Відповідь:

```
while True:
    try:
        number = int(input("Введіть число: "))
        print("Ви ввели число:", number)
        break
    except ValueError:
        print("Некоректне введення. Спробуйте ще раз.")
```

30. Що виведе наступний код?

```
for i in range(1, 11):
    if i % 2 == 0:
        print(f"{i} – парне число")
    else:
        print(f"{i} – непарне число")
```

Відповідь:

1 – непарне число

2 – парне число

3 – непарне число
4 – парне число
5 – непарне число
6 – парне число
7 – непарне число
8 – парне число
9 – непарне число
10 – парне число

Лекція №4. Функції у Python

4.1 Огляд концепції функцій

Огляд концепції функцій є важливим компонентом розуміння програмування на Python. Функції дозволяють організовувати код у модульні блоки, які можуть бути викликані повторно з різних частин програми. Ось огляд основних аспектів функцій у Python:

Визначення та Виклик Функцій

Функція – це блок коду, який виконується лише тоді, коли він викликається. Вона дозволяє організувати код у модульні, перевикористовувані частини. У Python функція визначається за допомогою ключового слова `def`, за яким слідує ім'я функції і круглі дужки.

```
def my_function():
  print("Hello, World!")
```

Щоб викликати цю функцію, потрібно просто використати її ім'я з дужками:

```
my_function()
```

Аргументи та Параметри

Функції можуть приймати аргументи, які дозволяють передавати дані у функцію. Параметри – це змінні, визначені у визначенні функції, а аргументи – це значення, які передаються функції під час її виклику.

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

Значення за Замовчуванням

Параметри функцій можуть мати значення за замовчуванням, які використовуються, якщо аргумент не був переданий під час виклику функції.

```
def greet(name="World"):
    print(f"Hello, {name}!")

greet()
greet("Alice")
```

Повернення Значення

Функції можуть повертати значення за допомогою ключового слова `return`. Це дозволяє функції відправляти результат своєї роботи назад до місця виклику.

```
 def add(a, b):
    return a + b

result = add(3, 5)
print(result)
```

Лямбда-функції

Python підтримує анонімні функції, відомі як лямбда-функції, які визначаються за допомогою ключового слова `lambda`. Вони можуть містити лише одну вираз і зазвичай використовуються для коротких функцій.

```
 add = lambda a, b: a + b
print(add(3, 5))
```

Функції як Об'єкти Першого Класу

У Python функції є об'єктами першого класу, що означає, що їх можна зберігати у змінних, передавати як аргументи іншим функціям і повертати з функцій.

```
def add(a, b):
   return a + b

def operate(func, x, y):
   return func(x, y)

print(operate(add, 3, 5))
```

Вкладені Функції та Області Видимості

Функції можуть бути визначені всередині інших функцій, і це призводить до створення вкладених областей видимості. Це дозволяє створювати замикання (closures), де вкладені функції зберігають доступ до змінних з області видимості, в якій вони були створені.

```
def outer_function(x):
   def inner_function(y):
      return x + y
```

```
    return inner_function

add_five = outer_function(5)
print(add_five(3))
```

Рекурсія

Функція може викликати саму себе, що відомо як рекурсія. Це корисно для вирішення завдань, які можуть бути розбиті на подібні підзавдання.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))
```

Використання Документування (Docstrings)

Python дозволяє документувати функції за допомогою спеціальних рядків документування, які можуть бути доступні через атрибут `__doc__`.

```
def greet(name):
    """Print a greeting to the user."""
    print(f"Hello, {name}!")

print(greet.__doc__)
```

Висновок

Функції є основним будівельним блоком в Python, що дозволяє розділити програму на модульні, перевикористовувані частини коду. Розуміння концепції функцій, таких як визначення, аргументи, значення за замовчуванням, повернення значень, лямбда-функції, вкладені функції, рекурсія та документування, є ключовим для ефективного програмування в Python.

4.2 Створення та виклик функцій

Створення та виклик функцій є основними елементами роботи з функціями у Python. Цей процес включає визначення функцій, їх виклик, а також роботу з аргументами та параметрами. Ось детальніший огляд:

Створення Функцій

Функції у Python створюються за допомогою ключового слова `def`, після якого слідує ім'я функції, круглі дужки і двокрапка. Тіло функції містить інструкції, які будуть виконуватися під час виклику функції, і має бути відступлене на один рівень.

Приклад простої функції:

```
def greet():
    print("Hello, World!")
```

Ця функція не приймає жодних аргументів і просто виводить повідомлення.

Аргументи та Параметри

Функції можуть приймати один або більше аргументів. Аргументи вказуються у круглих дужках під час визначення функції. Параметри – це змінні, що використовуються у функції для обробки даних.

Приклад функції з одним параметром:

```
def greet(name):
    print(f"Hello, {name}!")
```

Виклик Функцій

Щоб викликати функцію, потрібно написати її ім'я, після чого у круглих дужках вказати необхідні аргументи, якщо такі є.

Приклад виклику функції:

```
greet("Alice")
```

Значення за Замовчуванням

Параметри функцій можуть мати значення за замовчуванням, які використовуються, якщо аргумент не був переданий під час виклику функції.

Приклад функції зі значенням за замовчуванням:

```
def greet(name="World"):
    print(f"Hello, {name}!")
```

Виклики функції:

```
greet()          # Виведе: Hello, World!
greet("Alice")   # Виведе: Hello, Alice!
```

Повернення Значення

Функції можуть повертати значення за допомогою ключового слова `return`. Це дозволяє функції відправляти результат своєї роботи назад до місця виклику.

Приклад функції, яка повертає значення:

```
 def add(a, b):
    return a + b

result = add(3, 5)
print(result)  # Виведе: 8
```

Лямбда-Функції

Лямбда-функції – це анонімні функції, які визначаються за допомогою ключового слова `lambda`. Вони можуть мати будь-яку кількість аргументів, але лише один вираз.

Приклад лямбда-функції:

```
add = lambda a, b: a + b
print(add(3, 5))  # Виведе: 8
```

Вкладені Функції

Функції можуть бути визначені всередині інших функцій. Це призводить до створення вкладених областей видимості і дозволяє створювати замикання (closures).

Приклад вкладеної функції:

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

add_five = outer_function(5)
print(add_five(3))  # Виведе: 8
```

Виклик Функцій з Іншої Функції

Функції можуть викликати інші функції. Це дозволяє будувати більш складні програми з менших функціональних блоків.

Приклад виклику функції з іншої функції:

```
 def multiply(a, b):
    return a * b

def add_and_multiply(a, b, c):
    sum_result = add(a, b)
    return multiply(sum_result, c)

print(add_and_multiply(2, 3, 4))  # Виведе: 20
```

Висновок

Створення та виклик функцій у Python є базовими навичками, які дозволяють організувати код у зручні та перевикористовувані блоки. Розуміння концепцій визначення функцій, роботи з аргументами і параметрами, використання значень за замовчуванням, повернення значень, використання лямбда-функцій, вкладених функцій і виклику функцій з інших функцій є ключовими для ефективного програмування на Python.

4.3 Параметри та аргументи функцій

Параметри та аргументи функцій є однією з ключових концепцій у програмуванні на Python. Вони дозволяють передавати дані до функцій і впливають на поведінку функції під час її виконання. Ось детальний огляд цієї теми:

Параметри та Аргументи

- Параметри – це змінні, визначені у визначенні функції, які приймають значення під час виклику функції.
- Аргументи – це фактичні значення, передані функції під час її виклику.

Визначення Параметрів

Параметри вказуються у круглих дужках після імені функції у її визначенні. Функція може мати будь-яку кількість параметрів, включаючи жодного.

Приклад:

```
def greet(name):
    print(f"Hello, {name}!")
```

У цьому прикладі `name` – це параметр.

Передача Аргументів

Аргументи передаються функції під час її виклику, у тих же круглих дужках. Вони повинні відповідати параметрам за кількістю та порядком.

Приклад:

```
greet("Alice")
```

Тут `"Alice"` – це аргумент.

Типи Аргументів

#Позиційні Аргументи

Аргументи передаються функції в тому порядку, в якому вони визначені.

Приклад:

```
def add(a, b):
    return a + b

result = add(3, 5)  # 3 і 5 – позиційні аргументи
print(result)  # Виведе: 8
```

#Іменовані Аргументи

Аргументи передаються за ім'ям параметра. Це дозволяє змінювати порядок аргументів.

Приклад:

```
def greet(first_name, last_name):
    print(f"Hello, {first_name} {last_name}!")
greet(last_name="Doe", first_name="John")
```

#Значення за Замовчуванням

Параметри можуть мати значення за замовчуванням, які використовуються, якщо аргумент не переданий.

Приклад:

```
def greet(name="World"):
    print(f"Hello, {name}!")

greet()          # Виведе: Hello, World!
greet("Alice")   # Виведе: Hello, Alice!
```

#Аргументи з Невизначеною Кількістю (*args та **kwargs)

Python дозволяє передавати змінну кількість аргументів до функції за допомогою *args та **kwargs.

- `*args` дозволяє передавати змінну кількість позиційних аргументів.
- `**kwargs` дозволяє передавати змінну кількість іменованих аргументів.

Приклад з *args:

```
def add(*args):
    return sum(args)

print(add(1, 2, 3))  # Виведе: 6
```

Приклад з **kwargs:

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25)
```

Передача Аргументів за Посиланням

У Python аргументи передаються за посиланням, а не за значенням. Це означає, що зміни, внесені у змінну всередині функції, будуть відображені і зовні функції, якщо змінна є змінюваною (наприклад, список чи словник).

Приклад:

```
def update_list(my_list):
    my_list.append(4)

numbers = [1, 2, 3]
update_list(numbers)
print(numbers)  # Виведе: [1, 2, 3, 4]
```

Обмеження та Особливості

- Кількість переданих аргументів має відповідати кількості параметрів, якщо параметри не мають значень за замовчуванням.
- Порядок переданих позиційних аргументів важливий.
- Іменовані аргументи мають бути передані після позиційних.

Висновок

Параметри та аргументи функцій є важливими інструментами, які дозволяють гнучко і ефективно використовувати функції у Python. Розуміння різних типів аргументів і способів їх передачі дозволяє створювати більш адаптивні та модульні програми.

4.4 Повернення значень з функцій

Повернення значень з функцій є однією з основних концепцій у програмуванні на Python. Це дозволяє функціям обчислювати значення і передавати їх назад до точки виклику. Ось детальний огляд цієї теми:

Повернення Значень з Функцій

У Python функції можуть повертати значення за допомогою ключового слова `return`. Це дозволяє передавати результат виконання функції назад до місця її виклику.

Основи Використання `return`

Ключове слово `return` завершує виконання функції і повертає значення. Якщо `return` не вказано, функція повертає `None` за замовчуванням.

Приклад:

```
def add(a, b):
    return a + b

result = add(3, 5)
print(result)  # Виведе: 8
```

Одне і Багато Значень

Функція може повертати як одне значення, так і кілька значень одночасно. У Python кілька значень повертаються у вигляді кортежу.

#Повернення Одного Значення

Приклад:

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

result = square(4)
print(result)  # Виведе: 16
```

#Повернення Кількох Значень

Приклад:

```
def get_name_and_age():
    name = "Alice"
    age = 30
    return name, age
```

```
name, age = get_name_and_age()
print(name)  # Виведе: Alice
print(age)   # Виведе: 30
```

Використання `return` для Завершення Функції

Ключове слово `return` також може бути використано для завершення виконання функції до того, як вона досягне кінця. Це корисно для виходу з функції на основі умов.

Приклад:

```
def check_number(x):
    if x > 0:
        return "Positive"
    elif x < 0:
        return "Negative"
    return "Zero"

print(check_number(10))  # Виведе: Positive
print(check_number(-5))  # Виведе: Negative
print(check_number(0))   # Виведе: Zero
```

Повернення Значень у Лямбда-Функціях

Лямбда-функції автоматично повертають значення результату свого виразу, без використання `return`.

Приклад:

```
square = lambda x: x**2
```

```
print(square(4))  # Виведе: 16
```

Повернення Складних Об'єктів

Функції можуть повертати складні об'єкти, такі як списки, словники, інші функції або об'єкти класів.

Приклад:

```
def create_person(name, age):
    return {"name": name, "age": age}

person = create_person("Alice", 30)
print(person)  # Виведе: {'name': 'Alice', 'age': 30}
```

Повернення Замикань

Функція може повертати іншу функцію, створюючи замикання (closure). Це дозволяє створювати функції з внутрішнім станом.

Приклад:

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

add_five = outer_function(5)
print(add_five(3))  # Виведе: 8
```

Висновок

Повернення значень з функцій у Python дозволяє зробити код більш модульним та гнучким. Розуміння того, як використовувати ключове слово `return`, повертати одночасно кілька значень, працювати з лямбда-функціями і складними об'єктами, є важливими аспектами ефективного програмування на Python.

Питання для перевірки знань по темі лекції

1. Що таке функція у Python і чому вона використовується?

Функція у Python — це блок коду, який виконує певну задачу і може бути повторно використаний. Функції допомагають організувати код, зменшити його повторення та полегшити тестування і налагодження.

2. Як визначити функцію у Python? Напишіть приклад функції, яка не приймає аргументів і виводить "Hello, World!".

Функцію визначають за допомогою ключового слова `def`. Приклад:

```
def greet():
    print("Hello, World!")
```

3. Що таке аргументи та параметри функції? Чим вони відрізняються?

Параметри — це змінні, які визначаються у заголовку функції. Аргументи — це значення, які передаються функції при її виклику. Параметри описують, що функція очікує, а аргументи — це конкретні дані, які передаються в функцію.

4. Що таке значення за замовчуванням у параметрах функції? Напишіть приклад функції з параметром, який має значення за замовчуванням.

Значення за замовчуванням задає параметр функції, який використовується, якщо при виклику функції значення не передане. Приклад:

```
def greet(name="Guest"):
    print(f"Hello, {name}!")
```

5. Як функція може повертати значення? Напишіть приклад функції, яка приймає два аргументи і повертає їх суму.

Функція повертає значення за допомогою ключового слова `return`. Приклад:

```
def add(a, b):
    return a + b
```

6. Що таке лямбда-функція і як вона визначається у Python? Напишіть приклад лямбда-функції, яка повертає квадрат числа.

Лямбда-функція — це анонімна функція, яку можна визначити за допомогою ключового слова `lambda`. Приклад:

```
square = lambda x: x * x
```

7. Чи можуть функції бути вкладеними одна в одну у Python? Якщо так, наведіть приклад такої функції.

Так, функції можуть бути вкладеними. Приклад:

```
def outer():
    def inner():
        print("Inner function")
    inner()
```

8. Що таке рекурсія у контексті функцій? Напишіть приклад рекурсивної функції, яка обчислює факторіал числа.

Рекурсія — це процес, при якому функція викликає сама себе. Приклад рекурсивної функції для обчислення факторіалу:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

9. Як задокументувати функцію у Python за допомогою docstring? Напишіть приклад функції з docstring.

Для документування функцій використовують docstring, який розміщується між трійними лапками. Приклад:

```
def add(a, b):
    """
    Додає два числа і повертає результат.
    :param a: Перше число
    :param b: Друге число
    :return: Сума двох чисел
    """
    return a + b
```

10. Що таке замикання (closure) у Python і як воно працює? Наведіть приклад функції, яка повертає іншу функцію, створюючи замикання.

Замикання — це функція, яка повертає іншу функцію і має доступ до змінних з оточення. Приклад:

```
def outer(x):
    def inner(y):
```

```
        return x + y
    return inner

add_five = outer(5)
print(add_five(10))  # Виведе 15
```

11. Як визначити функцію у Python? Напишіть синтаксис визначення функції, яка не приймає аргументів.

Функція визначається за допомогою ключового слова `def`. Приклад:

```
def my_function():
    pass  # тіло функції може бути порожнім
```

12. Як викликати функцію у Python? Напишіть приклад виклику функції з аргументами.

Функцію викликають за її ім'ям з відповідними аргументами. Приклад:

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")  # виклик з аргументом
```

13. Що станеться, якщо викликати функцію без круглих дужок?

Якщо викликати функцію без круглих дужок, ви отримаєте посилання на саму функцію, але вона не виконається. Наприклад:

```
print(greet)  # виведе <function greet at 0x...>
```

14. Яка різниця між параметрами та аргументами у функціях?

Параметри — це змінні, визначені в оголошенні функції. Аргументи — це фактичні значення, які передаються функції при її виклику.

15. Як передати аргументи у функцію за іменем? Напишіть приклад такої функції і її виклику.

Аргументи можна передавати за іменем, використовуючи `key=value`. Приклад:

```
def introduce(name, age):
    print(f"Name: {name}, Age: {age}")

introduce(age=30, name="Bob")  # виклик з іменованими аргументами
```

16. Що таке значення за замовчуванням у параметрах функції? Напишіть приклад функції, що використовує значення за замовчуванням для одного з параметрів.

Значення за замовчуванням використовується, якщо аргумент не передається при виклику функції. Приклад:

```
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet()  # виведе "Hello, Guest!"
```

17. Як можна передати змінну кількість позиційних аргументів у функцію? Напишіть приклад такої функції з використанням `*args`.

Використовуйте `*args` для передачі змінної кількості аргументів. Приклад:

```
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3))  # виведе 6
```

18. Як можна передати змінну кількість іменованих аргументів у функцію? Напишіть приклад такої функції з використанням `**kwargs`.

Використовуйте `**kwargs` для передачі змінної кількості іменованих аргументів. Приклад:

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25)  # виведе "name: Alice", "age: 25"
```

19. Як можна викликати одну функцію всередині іншої функції? Напишіть приклад.

Одна функція може викликати іншу всередині свого тіла. Приклад:

```
def outer():
    def inner():
        return "Hello from inner"
    return inner()

print(outer())  # виведе "Hello from inner"
```

20. Що таке лямбда-функція у Python і як її викликати? Напишіть приклад визначення і виклику лямбда-функції.

Лямбда-функція — це коротка анонімна функція. Приклад:

```
square = lambda x: x * x
print(square(4))  # виведе 16
```

21. Чим відрізняються параметри та аргументи функцій у Python?

- Параметри — це змінні, визначені в оголошенні функції.
- Аргументи — це значення, які передаються при виклику функції.

22. Як визначити функцію з кількома параметрами? Напишіть приклад такої функції.

```
def add(a, b):
    return a + b
```

23. Що таке позиційні аргументи і як вони передаються у функцію? Напишіть приклад.

Позиційні аргументи передаються у функцію у порядку, в якому вони оголошені.

```
def multiply(x, y):
    return x * y

print(multiply(2, 3))  # 2 і 3 — позиційні аргументи
```

24. Що таке іменовані аргументи і як вони використовуються? Напишіть приклад функції з іменованими аргументами.

Іменовані аргументи передаються у функцію у форматі `key=value`.

```
def greet(name, age):
    print(f"Hello, {name}. You are {age} years old.")

greet(name="Alice", age=30)  # Іменовані аргументи
```

25. Як визначити параметри функції зі значеннями за замовчуванням? Напишіть приклад такої функції і її виклику.

Значення за замовчуванням задаються у визначенні функції.

```
def greet(name="Guest"):
    print(f"Hello, {name}!")
```

```
greet()        # Виведе: "Hello, Guest!"
greet("Alice")  # Виведе: "Hello, Alice!"
```

26. Що таке `*args` у функціях Python? Як його використовувати? Напишіть приклад функції, яка приймає змінну кількість позиційних аргументів.

`*args` дозволяє передавати змінну кількість позиційних аргументів.

```
def add_all(*args):
    return sum(args)

print(add_all(1, 2, 3, 4))  # Виведе 10
```

27. Що таке `**kwargs` у функціях Python? Як його використовувати? Напишіть приклад функції, яка приймає змінну кількість іменованих аргументів.

`kwargs` дозволяє передавати змінну кількість іменованих аргументів.

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=30)  # Виведе "name: Alice" і "age: 30"
```

28. Як передати список або словник як аргумент до функції, використовуючи розпаковку? Напишіть приклад для списку і словника.

Списки розпаковуються за допомогою `*`, а словники — за допомогою ``.

```
def sum_numbers(a, b, c):
    return a + b + c
```

```
numbers_list = [1, 2, 3]
print(sum_numbers(*numbers_list))  # Виведе 6

numbers_dict = {"a": 1, "b": 2, "c": 3}
print(sum_numbers(numbers_dict))  # Виведе 6
```

29. Як Python обробляє параметри зі змінними значеннями при передачі їх у функцію? Наведіть приклад з використанням списку або словника.

Python може змінювати вміст переданих об’єктів, якщо вони є змінними типами даних, наприклад списками чи словниками.

```
def add_item(my_list):
    my_list.append(4)

lst = [1, 2, 3]
add_item(lst)
print(lst)  # Виведе [1, 2, 3, 4] — список змінений
```

30. Чи можуть `*args` та `**kwargs` використовуватися разом у визначенні однієї функції? Якщо так, напишіть приклад такої функції і її виклику.

Так, `*args` і `**kwargs` можна використовувати разом.

```
def func_with_args_kwargs(*args, **kwargs):
    print("Args:", args)
    print("Kwargs:", **kwargs)

func_with_args_kwargs(1, 2, 3, name="Alice", age=30)
# Виведе:
# Args: (1, 2, 3)
```

```
# Kwargs: {'name': 'Alice', 'age': 30}
```

31. Що таке ключове слово `return` у Python і для чого воно використовується у функціях?

- Ключове слово `return` використовується для повернення результату з функції в точку її виклику.

32. Що станеться, якщо у функції не використати ключове слово `return`? Яке значення буде повернено?

- Якщо не використати `return`, функція поверне значення `None`.

33. Як функція може повертати більше одного значення? Напишіть приклад такої функції.

- Функція може повертати кілька значень через кортеж.

```
def get_coordinates():
    return 10, 20

x, y = get_coordinates()  # x = 10, y = 20
```

34. Чим відрізняється повернення значення з функції від виводу значення за допомогою `print`?

- `return` повертає значення з функції для подальшого використання, тоді як `print` просто виводить його на екран і нічого не повертає.

35. Як функція може повернути іншу функцію (замикання)? Напишіть приклад такої функції.

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function
```

```
add_five = outer_function(5)
print(add_five(10))  # Виведе 15
```

36. Що таке рекурсивна функція і як вона використовує `return`? Напишіть приклад рекурсивної функції, яка обчислює факторіал числа.

- Рекурсивна функція викликає сама себе до досягнення базового випадку.

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))  # Виведе 120
```

37. Як працює повернення значення у лямбда-функціях? Напишіть приклад лямбда-функції, яка повертає подвоєне значення аргументу.

- Лямбда-функції автоматично повертають результат обчислення без використання `return`.

```
double = lambda x: x * 2
print(double(5))  # Виведе 10
```

38. Як можна використати ключове слово `return` для виходу з функції до її завершення? Напишіть приклад функції, яка використовує `return` для завершення на основі умови.

```
def check_even(number):
    if number % 2 == 0:
        return "Even"
    return "Odd"
```

```
print(check_even(4))  # Виведе "Even"
```

39. Як функція може повертати складні об'єкти, такі як списки або словники? Напишіть приклад функції, яка повертає список чисел.

```
def get_numbers():
    return [1, 2, 3, 4, 5]

print(get_numbers())  # Виведе [1, 2, 3, 4, 5]
```

40. Чи можна використовувати `return` всередині циклу в функції? Якщо так, напишіть приклад такої функції.

- Так, `return` можна використовувати в циклі для виходу з функції.

```
def find_even(numbers):
    for num in numbers:
        if num % 2 == 0:
            return num
    return None

print(find_even([1, 3, 5, 6, 7]))  # Виведе 6
```

Лекція №5. Робота зі списками та кортежами

У Python, списки та кортежі є основними структурами даних, які використовуються для зберігання послідовностей елементів. Вони можуть містити дані різних типів, і мають свої особливості та методи для маніпулювання даними. У цій лекції ми розглянемо основні операції зі списками та кортежами, вбудовані функції для роботи зі списками, а також методи списків.

5.1 Операції зі списками та кортежами: індексація, зрізи

#Індексація

Індексація дозволяє отримувати доступ до окремих елементів списку чи кортежу. Індекси починаються з 0.

```
# Приклад зі списком
my_list = [10, 20, 30, 40, 50]
print(my_list[0])  # Виведе 10
print(my_list[2])  # Виведе 30

# Приклад з кортежем
my_tuple = (10, 20, 30, 40, 50)
print(my_tuple[0])  # Виведе 10
print(my_tuple[2])  # Виведе 30
```

#Зрізи

Зрізи дозволяють отримувати підпослідовності зі списків та кортежів. Формат зрізу: `sequence[start:stop:step]`.

```
 # Приклад зі списком
my_list = [10, 20, 30, 40, 50]
print(my_list[1:4])  # Виведе [20, 30, 40]
print(my_list[:3])   # Виведе [10, 20, 30]
print(my_list[::2])  # Виведе [10, 30, 50]

# Приклад з кортежем
my_tuple = (10, 20, 30, 40, 50)
```

```
print(my_tuple[1:4])  # Виведе (20, 30, 40)
print(my_tuple[:3])   # Виведе (10, 20, 30)
print(my_tuple[::2])  # Виведе (10, 30, 50)
```

2. Застосування вбудованих функцій для роботи зі списками

Python надає багато вбудованих функцій для роботи зі списками, таких як `len()`, `max()`, `min()`, `sum()`, `sorted()`, `list()`, та інші.

#`len()`

Функція `len()` повертає кількість елементів у списку або кортежі.

```
my_list = [10, 20, 30, 40, 50]
print(len(my_list))  # Виведе 5

my_tuple = (10, 20, 30, 40, 50)
print(len(my_tuple))  # Виведе 5
```

#`max()` і `min()`

Функції `max()` і `min()` повертають найбільший і найменший елемент відповідно.

```
my_list = [10, 20, 30, 40, 50]
print(max(my_list))  # Виведе 50
print(min(my_list))  # Виведе 10

my_tuple = (10, 20, 30, 40, 50)
print(max(my_tuple))  # Виведе 50
print(min(my_tuple))  # Виведе 10
```

#`sum()`

Функція `sum()` повертає суму всіх елементів у списку або кортежі.

```
my_list = [10, 20, 30, 40, 50]
print(sum(my_list))  # Виведе 150

my_tuple = (10, 20, 30, 40, 50)
print(sum(my_tuple))  # Виведе 150
```

#`sorted()`

Функція `sorted()` повертає новий відсортований список з елементами вхідного списку або кортежу.

```
my_list = [50, 20, 30, 10, 40]
print(sorted(my_list))  # Виведе [10, 20, 30, 40, 50]

my_tuple = (50, 20, 30, 10, 40)
print(sorted(my_tuple))  # Виведе [10, 20, 30, 40, 50]
```

3. Методи списків

Списки мають вбудовані методи, які дозволяють додавати, видаляти та маніпулювати елементами списку.

#`append()`

Метод `append()` додає елемент в кінець списку.

```
my_list = [10, 20, 30]
my_list.append(40)
print(my_list)  # Виведе [10, 20, 30, 40]
```

#`extend()`

Метод `extend()` додає всі елементи вказаного ітератора (списку, кортежу і т.д.) до кінця списку.

```
my_list = [10, 20, 30]
```

```
my_list.extend([40, 50])
print(my_list)  # Виведе [10, 20, 30, 40, 50]
```

#`insert()`

Метод `insert()` вставляє елемент у вказану позицію.

```
my_list = [10, 20, 30]
my_list.insert(1, 15)
print(my_list)  # Виведе [10, 15, 20, 30]
```

#`remove()`

Метод `remove()` видаляє перше входження вказаного елемента зі списку.

```
my_list = [10, 20, 30, 20]
my_list.remove(20)
print(my_list)  # Виведе [10, 30, 20]
```

#`pop()`

Метод `pop()` видаляє та повертає елемент зі списку за вказаним індексом. Якщо індекс не вказано, видаляє та повертає останній елемент.

```
my_list = [10, 20, 30]
print(my_list.pop())  # Виведе 30
print(my_list)        # Виведе [10, 20]

print(my_list.pop(0))  # Виведе 10
print(my_list)         # Виведе [20]
```

#`clear()`

Метод `clear()` видаляє всі елементи зі списку.

```
my_list = [10, 20, 30]
my_list.clear()
print(my_list)  # Виведе []
```

#`index()`

Метод `index()` повертає індекс першого входження вказаного елемента.

```
my_list = [10, 20, 30, 20]
print(my_list.index(20))  # Виведе 1
```

#`count()`

Метод `count()` повертає кількість входжень вказаного елемента.

```
my_list = [10, 20, 30, 20]
print(my_list.count(20))  # Виведе 2
```

#`sort()`

Метод `sort()` сортує елементи списку на місці.

```
my_list = [50, 20, 30, 10, 40]
my_list.sort()
print(my_list)  # Виведе [10, 20, 30, 40, 50]
```

#`reverse()`

Метод `reverse()` змінює порядок елементів списку на зворотний.

```
my_list = [10, 20, 30]
my_list.reverse()
```

```
print(my_list)  # Виведе [30, 20, 10]
```

Генератори списків у Python дозволяють створювати нові списки з наявних, застосовуючи певну логіку в компактній та зручній формі. Це зручний спосіб скорочено записувати цикли для заповнення списків.

Синтаксис генератора списку:

```
[expression for item in iterable if condition]
```

- `expression`: вираз, який буде обчислюватися для кожного елемента.
- `item`: кожен елемент з ітерабельного об'єкта (список, рядок, тощо).
- `iterable`: ітерабельний об'єкт (наприклад, список, рядок, кортеж).
- `condition` (необов'язковий): умова для фільтрації елементів.

Приклади генераторів списків:

1. Створення списку квадратів чисел від 0 до 9:

```
squares = [x2 for x in range(10)]
print(squares)
```

Виведе:

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

2. Фільтрація та перетворення: парні числа з діапазону від 0 до 9:

```
evens = [x for x in range(10) if x % 2 == 0]
print(evens)
```

Виведе:

```
[0, 2, 4, 6, 8]
```

3. Конвертація рядків у списку до великих літер:

```
words = ["hello", "world", "python"]
uppercase_words = [word.upper() for word in words]
print(uppercase_words)
```

Виведе:

['HELLO', 'WORLD', 'PYTHON']

4. Комбінація елементів з двох списків:

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
combined = [(x, y) for x in list1 for y in list2]
print(combined)
```

Виведе:

[(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (3, 'a'), (3, 'b'), (3, 'c')]

Генератори списків з умовою:
Якщо потрібно включати елементи лише за певної умови, можна використовувати `if` всередині генератора.

5. Список квадратів парних чисел:

```
even_squares = [x2 for x in range(10) if x % 2 == 0]
print(even_squares)
```

Виведе:

[0, 4, 16, 36, 64]

Генератори вкладених списків:
Для роботи з багатовимірними списками або вкладеними циклами.

6. Розпакування двовимірного списку:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat_list = [num for row in matrix for num in row]
print(flat_list)
```

Виведе:

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

Генератори списків є потужним інструментом для роботи зі списками у Python, дозволяючи компактно та ефективно створювати нові списки.

Висновок

Списки та кортежі є потужними структурами даних у Python, які дозволяють ефективно зберігати та маніпулювати даними. Використання індексації, зрізів, вбудованих функцій та методів списків значно спрощує роботу з цими структурами. Надіюся, що цей огляд допоможе вам краще розуміти та використовувати списки та кортежі у ваших Python-програмах.

Питання для перевірки знань по темі лекції

1. Що таке індексація в контексті списків та кортежів у Python?
– Індексація дозволяє отримати доступ до окремих елементів списку або кортежу за допомогою індексу. Індекси починаються з 0.

2. Як можна отримати перший елемент списку `my_list = [5, 10, 15, 20]` за допомогою індексації?
– Використовуючи `my_list[0]`.

3. Що поверне вираз `my_tuple[-1]`, якщо `my_tuple = (1, 2, 3, 4, 5)`?
– Вираз поверне останній елемент кортежу, тобто 5.

4. Як можна отримати підпослідовність [2, 3, 4] зі списку `numbers = [1, 2, 3, 4, 5]` за допомогою зрізу?
– Використовуючи `numbers[1:4]`.

5. Чим відрізняються списки та кортежі в Python?
– Списки є змінюваними, тоді як кортежі є незмінюваними.

6. Що поверне вираз `my_list[::2]`, якщо `my_list = [10, 20, 30, 40, 50]`?
– Вираз поверне кожний другий елемент списку, тобто [10, 30, 50].

7. Як за допомогою зрізу отримати останні три елементи списку `my_list = [1, 2, 3, 4, 5, 6]`?
– Використовуючи `my_list[-3:]`.

8. Як створити зріз, що повертає елементи списку у зворотному порядку, якщо `my_list = [1, 2, 3, 4, 5]`?

– Використовуючи `my_list[::-1]`.

9. Що станеться, якщо спробувати отримати елемент за індексом, який перевищує довжину списку, наприклад, `my_list[10]`, якщо `my_list = [1, 2, 3]`?
– Виникне помилка `IndexError: list index out of range`.

10. Як можна за допомогою індексації отримати передостанній елемент кортежу `my_tuple = (10, 20, 30, 40, 50)`?
– Використовуючи `my_tuple[-2]`.

1. Яка вбудована функція використовується для визначення кількості елементів у списку або кортежі?
– Функція `len()`.

2. Як за допомогою функції `max()` знайти найбільший елемент у списку `numbers = [1, 3, 5, 7, 9]`?
– Використовуючи `max(numbers)`.

3. Яка вбудована функція повертає найменший елемент у списку?
– Функція `min()`.

4. Як за допомогою функції `sum()` знайти суму всіх елементів у списку `my_list = [10, 20, 30]`?
– Використовуючи `sum(my_list)`.

5. Що робить функція `sorted()` у Python?
– Функція `sorted()` повертає новий відсортований список з елементами вхідного списку або кортежу.

6. Як за допомогою функції `sorted()` відсортувати список `my_list = [40, 10, 30, 20]`?
– Використовуючи `sorted(my_list)`.

7. Яка різниця між функцією `sorted()` та методом `sort()` для списків у Python?
– `sorted()` повертає новий відсортований список, а `sort()` сортує список на місці і нічого не повертає.

8. Як за допомогою функції `list()` перетворити кортеж `my_tuple = (1, 2, 3)` на список?
– Використовуючи `list(my_tuple)`.

9. Що поверне вираз `len([])`?
– Вираз поверне 0, оскільки список порожній.

10. Яка вбудована функція використовується для створення списку з ітератора, наприклад, з рядка `s = "hello"`?
– Функція `list()`, наприклад, `list(s)` поверне `['h', 'e', 'l', 'l', 'o']`.

Перевірка знань: Методи списків

1. Який метод списку використовується для додавання елемента в кінець списку?
– Метод `append()`.

2. Як за допомогою методу `extend()` додати всі елементи списку `new_items = [40, 50]` до списку `my_list = [10, 20, 30]`?
– Використовуючи `my_list.extend(new_items)`.

3. Який метод списку дозволяє вставити елемент на конкретну позицію в списку?
– Метод `insert()`.

4. Що робить метод `remove()` для списків у Python?
– Метод `remove()` видаляє перше входження вказаного елемента зі списку.

5. Як за допомогою методу `pop()` видалити та повернути останній елемент зі списку `my_list = [10, 20, 30]`?
– Використовуючи `my_list.pop()`.

6. Який метод списку видаляє всі елементи зі списку?
– Метод `clear()`.

7. Як знайти індекс першого входження елемента 20 у списку `my_list = [10, 20, 30, 20]` за допомогою методу?
– Використовуючи `my_list.index(20)`.

8. Що поверне вираз `my_list.count(20)` для списку `my_list = [10, 20, 30, 20, 20]`?
– Вираз поверне 3, оскільки елемент 20 входить у список тричі.

9. Як відсортувати елементи списку `my_list = [50, 20, 30, 10, 40]` на місці за допомогою методу?
– Використовуючи `my_list.sort()`.

10. Який метод списку змінює порядок елементів на зворотний?
– Метод `reverse()`.

Лекція №6. Робота з рядками

Рядки в Python є одним з найважливіших типів даних. Вони використовуються для зберігання та маніпулювання текстовою інформацією. Рядки є незмінними, що означає, що їх не можна змінювати після створення, але можна створювати нові рядки шляхом обробки існуючих.

Операції з рядками

Конкатенація
Конкатенація – це процес об'єднання двох або більше рядків в один.

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)  # Виведе: Hello World
```

Індексація
Індексація дозволяє доступ до окремих символів у рядку. Індексація починається з нуля.

```
text = "Python"
first_char = text[0]
last_char = text[-1]
print(first_char)  # Виведе: P
print(last_char)   # Виведе: n
```

Зрізи
Зрізи дозволяють отримувати підрядки з рядка за допомогою вказівки діапазону індексів.

```
text = "Python Programming"
substring1 = text[0:6]
substring2 = text[7:]
print(substring1)  # Виведе: Python
print(substring2)  # Виведе: Programming
```

Застосування методів рядків

Python надає безліч вбудованих методів для роботи з рядками. Розглянемо деякі з них:

Метод `upper()`
Метод `upper()` перетворює всі символи рядка на великі літери.

```
text = "hello"
uppercase_text = text.upper()
print(uppercase_text)  # Виведе: HELLO
```

Метод `lower()`
Метод `lower()` перетворює всі символи рядка на малі літери.

```
text = "HELLO"
lowercase_text = text.lower()
print(lowercase_text)  # Виведе: hello
```

Метод `strip()`
Метод `strip()` видаляє пробіли з початку і кінця рядка.

```
text = "   Hello World   "
stripped_text = text.strip()
print(stripped_text)  # Виведе: Hello World
```

Метод `replace()`
Метод `replace()` замінює підрядок у рядку іншим підрядком.

```
text = "Hello World"
replaced_text = text.replace("World", "Python")
print(replaced_text)  # Виведе: Hello Python
```

Метод `split()`
Метод `split()` розбиває рядок на список підрядків за вказаним роздільником.

```
text = "apple,banana,cherry"
fruits = text.split(",")
print(fruits)  # Виведе: ['apple', 'banana', 'cherry']
```

Метод `join()`
Метод `join()` об'єднує елементи списку в один рядок з вказаним роздільником.

```
fruits = ['apple', 'banana', 'cherry']
text = ", ".join(fruits)
print(text)  # Виведе: apple, banana, cherry
```

Форматування рядків

Форматування рядків дозволяє вставляти значення змінних у рядки.

Використання оператора `%`

```
name = "Alice"
age = 30
text = "My name is %s and I am %d years old" % (name, age)
print(text)  # Виведе: My name is Alice and I am 30 years old
```

Метод `format()`

```
name = "Alice"
age = 30
text = "My name is {} and I am {} years old".format(name, age)
print(text)  # Виведе: My name is Alice and I am 30 years old
```

f-рядки (f-strings)

```
name = "Alice"
age = 30
text = f"My name is {name} and I am {age} years old"
print(text)  # Виведе: My name is Alice and I am 30 years old
```

Висновок

Рядки є невід'ємною частиною програмування на Python. Операції з рядками, методи рядків та форматування рядків забезпечують потужні інструменти для роботи з текстовими даними. Вивчення та практичне застосування цих можливостей дозволяє ефективно обробляти та маніпулювати текстовою інформацією.

Питання для перевірки знань по темі лекції

Питання для перевірки знань: Операції з рядками (Конкатенація, Індексація, Зрізи)

1. Що таке конкатенація рядків у Python? Як її можна виконати?
– Відповідь: Конкатенація рядків – це процес об'єднання двох або більше рядків в один. Це можна зробити за допомогою оператора `+`. Наприклад: `result = "Hello" + " " + "World"`

2. Як отримати перший символ рядка в Python?
– Відповідь: Для отримання першого символа рядка можна використати індексацію з індексом 0. Наприклад: `first_char = "Hello"[0]`

3. Що буде результатом виконання наступного коду: `text = "Python"; char = text[-1]`?
– Відповідь: Результатом буде останній символ рядка "Python", тобто `'n'`.

4. Як створити підрядок з першого по третій символи рядка `text = "Programming"`?
– Відповідь: Для створення підрядка можна використати зрізи з індексами 0 до 3. Наприклад: `substring = text[0:3]`, результат буде `'Pro'`.

5. Що поверне наступний код: `text = "Hello World"; sub = text[6:]`?
– Відповідь: Код поверне підрядок від 6-го символа до кінця рядка, тобто `'World'`.

6. Як отримати підрядок, що містить перші 5 символів рядка `text = "Data Science"`?
– Відповідь: Для отримання підрядка можна використати зріз з індексом 0 до 5. Наприклад: `substring = text[:5]`, результат буде `'Data '`.

7. Що буде результатом виконання наступного коду: `text1 = "Data"; text2 = "Science"; result = text1 + text2`?
– Відповідь: Результатом буде рядок `'DataScience'`.

8. Як можна отримати кожен другий символ рядка `text = "Python"`?
– Відповідь: Можна використати зріз з кроком 2. Наприклад: `substring = text[::2]`, результат буде `'Pto'`.

9. Що буде результатом виконання наступного коду: `text = "Hello World"; char = text[11]`?
– Відповідь: Код викличе помилку `IndexError`, оскільки індекс 11 виходить за межі рядка "Hello World", який має довжину 11.

10. Як отримати підрядок, що містить останні 4 символи рядка `text = "Machine Learning"`?
– Відповідь: Для отримання підрядка можна використати негативну індексацію. Наприклад: `substring = text[-4:]`, результат буде `'ning'`.

Питання для перевірки знань: Застосування методів рядків

1. Що робить метод `upper()` у Python? Наведіть приклад.
– Відповідь: Метод `upper()` перетворює всі символи рядка на великі літери. Наприклад: `"hello".upper()` поверне `'HELLO'`.

2. Як працює метод `lower()`? Покажіть на прикладі.
– Відповідь: Метод `lower()` перетворює всі символи рядка на малі літери. Наприклад: `"HELLO".lower()` поверне `'hello'`.

3. Що робить метод `strip()`? Як видалити пробіли з початку та кінця рядка?
– Відповідь: Метод `strip()` видаляє пробіли з початку і кінця рядка. Наприклад: `"  Hello  ".strip()` поверне `'Hello'`.

4. Яким методом можна замінити всі входження одного підрядка на інший? Наведіть приклад.
– Відповідь: Для заміни всіх входжень одного підрядка на інший використовується метод `replace()`. Наприклад: `"Hello World".replace("World", "Python")` поверне `'Hello Python'`.

5. Як розбити рядок на список підрядків за допомогою певного роздільника? Покажіть приклад з роздільником кома.

– Відповідь: Для розбиття рядка на список підрядків за допомогою роздільника використовується метод `split()`. Наприклад: `"apple,banana,cherry".split(",")` поверне `['apple', 'banana', 'cherry']`.

6. Що робить метод `join()`? Наведіть приклад об'єднання списку рядків у один рядок з роздільником пробіл.
– Відповідь: Метод `join()` об'єднує елементи списку в один рядок з вказаним роздільником. Наприклад: `["apple", "banana", "cherry"].join(" ")` поверне ``apple banana cherry`.

7. Як перевірити, чи містить рядок лише цифри? Який метод для цього використовується?
– Відповідь: Для перевірки, чи містить рядок лише цифри, використовується метод `isdigit()`. Наприклад: `"12345".isdigit()` поверне `True`, а `"123a5".isdigit()` поверне `False`.

8. Що робить метод `startswith()`? Покажіть приклад перевірки, чи починається рядок з певного підрядка.
– Відповідь: Метод `startswith()` перевіряє, чи починається рядок з певного підрядка. Наприклад: `"Hello World".startswith("Hello")` поверне `True`.

9. Як перевірити, чи закінчується рядок певним підрядком? Наведіть приклад використання методу `endswith()`.
– Відповідь: Метод `endswith()` перевіряє, чи закінчується рядок певним підрядком. Наприклад: `"Hello World".endswith("World")` поверне `True`.

10. Що робить метод `find()`? Як знайти індекс першого входження підрядка в рядку?
– Відповідь: Метод `find()` знаходить індекс першого входження підрядка в рядку. Якщо підрядок не знайдено, повертається `-1`. Наприклад: `"Hello World".find("World")` поверне `6`.

Питання для перевірки знань: Форматування рядків

1. Що таке форматування рядків у Python? Навіщо воно використовується?
– Відповідь: Форматування рядків у Python використовується для вставки значень змінних у рядки. Це дозволяє створювати динамічні текстові повідомлення з змінними даними.

2. Як працює оператор `%` для форматування рядків? Наведіть приклад.

– Відповідь: Оператор `%` використовується для вставки значень змінних у рядок з використанням специфікаторів форматування. Наприклад: `name = "Alice"; age = 30; text = "My name is %s and I am %d years old" % (name, age)` поверне ``My name is Alice and I am 30 years old``.

3. Як використовувати метод `format()` для форматування рядків? Наведіть приклад.

– Відповідь: Метод `format()` дозволяє вставляти значення змінних у рядок за допомогою фігурних дужок `{}`. Наприклад: `name = "Alice"; age = 30; text = "My name is {} and I am {} years old".format(name, age)` поверне ``My name is Alice and I am 30 years old``.

4. Що таке f-рядки (f-strings) у Python і як їх використовувати? Наведіть приклад.

– Відповідь: F-рядки дозволяють вставляти значення змінних безпосередньо в рядок, використовуючи префікс `f` перед рядком. Наприклад: `name = "Alice"; age = 30; text = f"My name is {name} and I am {age} years old"` поверне ``My name is Alice and I am 30 years old``.

5. Як можна задати точність для чисел з плаваючою точкою при форматуванні рядків? Наведіть приклад.

– Відповідь: Точність для чисел з плаваючою точкою можна задати за допомогою специфікатора `.nf` або в методі `format()` як `{:.nf}`. Наприклад: `value = 3.14159; formatted_value = "{:.2f}".format(value)` поверне ``3.14``.

6. Що буде результатом наступного коду: `"{:>10}".format("test")`?

– Відповідь: Результатом буде рядок ``      test``, де слово "test" вирівняне по правому краю з шириною поля 10 символів.

7. Як вставити значення змінної в рядок кілька разів, використовуючи метод `format()`? Наведіть приклад.

– Відповідь: Значення змінної можна вставити кілька разів, повторюючи відповідний вираз у рядку. Наприклад: `value = 42; text = "Value is {0}, really {0}".format(value)` поверне ``Value is 42, really 42``.

8. Як можна задати значення змінних за іменем у методі `format()`? Наведіть приклад.

– Відповідь: Значення змінних можна задавати за іменем, використовуючи формат `{name}` у рядку і передаючи аргументи у вигляді іменованих параметрів. Наприклад: `text = "My name is {name} and I am {age} years old".format(name="Alice", age=30)` поверне `'My name is Alice and I am 30 years old'`.

9. Що буде результатом наступного коду: `f"{{}} {5*7}"`?
– Відповідь: Результатом буде рядок `'{} 35'`, де `{{}}` використовується для відображення фігурних дужок у f-рядку.

10. Як можна використати метод `format()` для вирівнювання рядків по центру? Наведіть приклад.
– Відповідь: Для вирівнювання рядків по центру використовується специфікатор `^` з шириною поля. Наприклад: `"{:^10}".format("test")` поверне `' test '`.

Лекція №7. Робота зі словниками

Визначення словників та їх застосування

Словники – це колекція пар "ключ-значення", де ключі є унікальними, а значення можуть бути будь-якого типу даних. Вони схожі на списки, але замість індексів використовуються ключі.

#Приклад створення словника:

```
student = {
   "name": "John",
   "age": 21,
   "courses": ["Math", "CompSci"]
}
print(student)
```

#Застосування словників:
1. Зберігання конфігурацій: Конфігураційні налаштування програм часто зберігаються у словниках.
2. Збереження інформації про об'єкти: Наприклад, інформація про студентів у базі даних.
3. Підрахунок частоти: Наприклад, підрахунок кількості появ слів у тексті.

Операції зі словниками: доступ до елементів, додавання, видалення

#Доступ до елементів:
Щоб отримати значення за ключем, використовуємо квадратні дужки або метод `get`.

```
# Доступ за ключем
print(student["name"])  # Виведе: John

# Використання get
print(student.get("age"))  # Виведе: 21
```

#Додавання та зміна елементів:
Для додавання нового елемента або зміни існуючого використовуємо квадратні дужки з ключем.

```
# Додавання нового ключа-значення
student["grade"] = "A"

# Зміна значення існуючого ключа
student["age"] = 22
print(student)
```

#Видалення елементів:
Для видалення елемента за ключем використовуємо ключове слово `del` або метод `pop`.

```
# Видалення за допомогою del
del student["courses"]

# Видалення за допомогою pop
age = student.pop("age")
print(student)
print(age)  # Виведе: 22
```

Методи словників

Словники мають багато корисних методів для маніпуляції даними.

#Метод `keys`:
Повертає всі ключі словника.

```
keys = student.keys()
print(keys)  # Виведе: dict_keys(['name', 'grade'])
```

#Метод `values`:
Повертає всі значення словника.

```
values = student.values()
print(values)  # Виведе: dict_values(['John', 'A'])
```

#Метод `items`:
Повертає всі пари "ключ-значення" як кортежі.

```
items = student.items()
print(items)  # Виведе: dict_items([('name', 'John'), ('grade', 'A')])
```

#Метод `update`:
Оновлює словник іншим словником або ітерованим об'єктом пар "ключ-значення".

```
new_info = {"age": 23, "courses": ["Math", "CompSci"]}
student.update(new_info)
print(student)  # Виведе: {'name': 'John', 'grade': 'A', 'age': 23, 'courses': ['Math', 'CompSci']}
```

#Метод `clear`:
Очищує словник, видаляючи всі елементи.

```
student.clear()
print(student)  # Виведе: {}
```

Підсумок

Словники є потужним засобом для зберігання та маніпуляції даними у вигляді пар "ключ-значення". Вони забезпечують ефективний доступ до даних за ключами, надають можливості для додавання, видалення та зміни елементів, а також мають вбудовані методи для різних операцій. Володіння словниками значно спрощує роботу з даними та їх обробку у Python.

Питання для перевірки знань по темі лекції

10 питань для перевірки знань на тему "Визначення словників та їх застосування" з короткими відповідями:

1. Що таке словник у Python?
   – Словник — це структура даних, яка зберігає пари "ключ-значення". Ключі мають бути унікальними та незмінними.

2. Як створити словник?
   – Словник створюється за допомогою фігурних дужок: `my_dict = {}` або функції `dict()`.

3. Як додати новий елемент у словник?
   – Використовуючи синтаксис: `my_dict['new_key'] = value`.

4. Як отримати значення з словника за ключем?
   – Використовуючи ключ: `value = my_dict['key']`.

5. Що відбудеться, якщо спробувати отримати значення за неіснуючим ключем?
   – Виникне помилка `KeyError`.

6. Як перевірити наявність ключа у словнику?
– Використовуючи оператор `in`: `'key' in my_dict`.

7. Як видалити елемент зі словника?
– Використовуючи метод `pop()`: `my_dict.pop('key')` або оператор `del`: `del my_dict['key']`.

8. Як об'єднати два словники?
– Використовуючи метод `update()`: `my_dict.update(other_dict)` або оператор ``: `new_dict = {dict1, dict2}`.

9. Як отримати всі ключі словника?
– Використовуючи метод `keys()`: `keys = my_dict.keys()`.

10. Як ітеруватися по парам "ключ-значення" у словнику?
– Використовуючи метод `items()`: `for key, value in my_dict.items():`.

10 питань для перевірки знань на тему "Операції зі словниками: доступ до елементів, додавання, видалення" з короткими відповідями:

1. Як отримати значення з словника за ключем?
– Використовуючи синтаксис: `value = my_dict['key']`.

2. Як отримати значення з словника за ключем без виникнення помилки, якщо ключ відсутній?
– Використовуючи метод `get()`: `value = my_dict.get('key')`.

3. Як перевірити наявність ключа у словнику?
– Використовуючи оператор `in`: `'key' in my_dict`.

4. Як додати новий елемент до словника?
– Використовуючи синтаксис: `my_dict['new_key'] = value`.

5. Як оновити значення існуючого ключа у словнику?
– Використовуючи синтаксис: `my_dict['existing_key'] = new_value`.

6. Як видалити елемент зі словника за ключем?
– Використовуючи метод `pop()`: `value = my_dict.pop('key')` або оператор `del`: `del my_dict['key']`.

7. Що робить метод `popitem()` у словнику?

– Видаляє та повертає останню пару "ключ-значення".

8. Як очистити словник від усіх елементів?
– Використовуючи метод `clear()`: `my_dict.clear()`.

9. Як отримати всі ключі словника?
– Використовуючи метод `keys()`: `keys = my_dict.keys()`.

10. Як отримати всі значення словника?
– Використовуючи метод `values()`: `values = my_dict.values()`.

10 питань для перевірки знань на тему "Методи словників" з короткими відповідями:

1. Як працює метод `get()` у словнику?
– Повертає значення за вказаним ключем, або значення за замовчуванням, якщо ключ відсутній: `value = my_dict.get('key', default_value)`.

2. Що робить метод `keys()` у словнику?
– Повертає об'єкт-вид, який містить всі ключі словника: `keys = my_dict.keys()`.

3. Що робить метод `values()` у словнику?
– Повертає об'єкт-вид, який містить всі значення словника: `values = my_dict.values()`.

4. Як працює метод `items()` у словнику?
– Повертає об'єкт-вид, який містить всі пари "ключ-значення": `items = my_dict.items()`.

5. Що робить метод `update()` у словнику?
– Оновлює словник парами "ключ-значення" з іншого словника або ітерабельного об'єкта: `my_dict.update(other_dict)`.

6. Як працює метод `pop()` у словнику?
– Видаляє елемент за ключем і повертає його значення, або значення за замовчуванням, якщо ключ відсутній: `value = my_dict.pop('key', default_value)`.

7. Що робить метод `popitem()` у словнику?

– Видаляє та повертає останню пару "ключ-значення" зі словника: `key, value = my_dict.popitem()`.

8. Як працює метод `setdefault()` у словнику?
– Повертає значення за ключем, якщо ключ присутній, інакше додає ключ із зазначеним значенням і повертає це значення: `value = my_dict.setdefault('key', default_value)`.

9. Що робить метод `clear()` у словнику?
– Видаляє всі елементи зі словника: `my_dict.clear()`.

10. Як створити копію словника за допомогою методу `copy()`?
– Повертає поверхневу копію словника: `new_dict = my_dict.copy()`.

Лекція №8. Робота з файлами

Робота з файлами є важливою складовою програмування, яка дозволяє зберігати та отримувати дані. У Python є потужні вбудовані можливості для роботи з файлами, які ми детально розглянемо у цій лекції.

Відкриття та закриття файлів

Відкриття файлу:
Щоб відкрити файл, використовується функція `open()`, яка повертає файловий об'єкт. Формат відкриття файлу:

```
file_object = open('filename', 'mode')
```

Де `filename` — це ім'я файлу, а `mode` — це режим відкриття файлу:
- `'r'` — читання (за замовчуванням)
- `'w'` — запис (створює новий файл або перезаписує існуючий)
- `'a'` — додавання до існуючого файлу
- `'b'` — двійковий режим
- `'t'` — текстовий режим (за замовчуванням)
- `'+'` — оновлення (читання і запис)

Закриття файлу:
Після завершення роботи з файлом його потрібно закрити, щоб зберегти зміни та звільнити ресурси:

```
file_object.close()
```

Приклад:

```
file = open('example.txt', 'r')
```

```
# Виконання операцій з файлом
file.close()
```

Зчитування та запис даних у файл

Зчитування даних:
Для зчитування даних з файлу використовуються методи `read()`, `readline()`, та `readlines()`.

- `read(size)` — зчитує `size` символів (за замовчуванням зчитує весь файл)
- `readline()` — зчитує один рядок з файлу
- `readlines()` — зчитує всі рядки та повертає список рядків

Приклад:

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

Запис даних:
Для запису даних у файл використовуються методи `write()` та `writelines()`.

- `write(string)` — записує рядок у файл
- `writelines(list)` — записує список рядків у файл

Приклад:

```
with open('example.txt', 'w') as file:
    file.write("Hello, world!")
```

Робота з різними типами файлів

Текстові файли:
Текстові файли зберігають дані у вигляді тексту. Вони зручні для зберігання та обробки текстової інформації.

Приклад:

```
with open('text_file.txt', 'w') as file:
    file.write("Це приклад текстового файлу.")
```

Бінарні файли:
Бінарні файли зберігають дані у вигляді байтів. Вони використовуються для зберігання даних, які не є текстовими, таких як зображення, відео, аудіо, та інші файли.

Приклад:

```
with open('binary_file.bin', 'wb') as file:
    file.write(b'\x00\x01\x02\x03\x04')
```

Висновок
Робота з файлами у Python є надзвичайно гнучкою і дозволяє ефективно зберігати та отримувати дані. Важливо розуміти, як правильно відкривати, закривати, зчитувати та записувати дані у файли, а також працювати з різними типами файлів залежно від вимог вашого проекту.

Питання для перевірки знань по темі лекції

1. Як відкрити файл для читання у Python?
 – Для відкриття файлу для читання у Python використовується функція `open('filename', 'r')`.

2. Що повертає функція `open()`?
– Функція `open()` повертає файловий об'єкт.

3. Який метод використовується для закриття файлу?
– Для закриття файлу використовується метод `close()`, як у `file.close()`.

4. Який режим відкриття файлу використовується для запису, що перезаписує існуючий файл?
– Для запису, що перезаписує існуючий файл, використовується режим `'w'`.

5. Що відбувається, якщо спробувати відкрити неіснуючий файл у режимі читання (`'r'`)?
– Якщо спробувати відкрити неіснуючий файл у режимі читання (`'r'`), виникне помилка `FileNotFoundError`.

6. Яка конструкція забезпечує автоматичне закриття файлу після завершення роботи з ним?
– Для автоматичного закриття файлу після завершення роботи з ним використовується конструкція `with`, як у `with open('filename', 'r') as file:`.

7. Який режим відкриття файлу використовується для додавання інформації до існуючого файлу?
– Для додавання інформації до існуючого файлу використовується режим `'a'`.

8. Що відбувається, якщо відкрити файл у режимі `'w'`, якщо файл вже існує?
– Якщо відкрити файл у режимі `'w'`, і файл вже існує, він буде перезаписаний.

9. Який режим відкриття файлу використовується для двійкового читання?
– Для двійкового читання використовується режим `'rb'`.

10. Що означає режим `'+'` при відкритті файлу?
– Режим `'+'` при відкритті файлу означає оновлення, тобто читання і запис. Наприклад, `'r+'` відкриває файл для читання і запису.

1. Як прочитати весь вміст файлу в одну змінну?
– Для зчитування всього вмісту файлу в одну змінну використовується метод `read()`, як у `content = file.read()`.

2. Як зчитати один рядок з файлу?
– Для зчитування одного рядка з файлу використовується метод `readline()`, як у `line = file.readline()`.

3. Як зчитати всі рядки файлу у список?
– Для зчитування всіх рядків файлу у список використовується метод `readlines()`, як у `lines = file.readlines()`.

4. Як записати рядок у файл?
– Для запису рядка у файл використовується метод `write()`, як у `file.write('Hello, world!')`.

5. Як записати список рядків у файл?
– Для запису списку рядків у файл використовується метод `writelines()`, як у `file.writelines(['Line 1\n', 'Line 2\n'])`.

6. Що потрібно зробити, щоб записувати дані у файл по одному рядку?
– Щоб записувати дані у файл по одному рядку, можна використовувати цикл, як у:

```
with open('file.txt', 'w') as file:
    for line in lines:
        file.write(line)
```

7. Як записати бінарні дані у файл?
– Для запису бінарних даних у файл використовується метод `write()` в режимі `'wb'`, як у `file.write(b'\x00\x01\x02\x03')`.

8. Як додати текст до існуючого файлу?
– Щоб додати текст до існуючого файлу, потрібно відкрити файл у режимі `'a'`, як у `with open('file.txt', 'a') as file:`.

9. Як прочитати файл построчно у циклі?
– Для построчного зчитування файлу у циклі можна використовувати:

```
with open('file.txt', 'r') as file:
```

```
for line in file:
    print(line)
```

10. Що станеться, якщо використовувати `read()` без параметра в режимі обробки великих файлів?
– Використання `read()` без параметра може призвести до завантаження всього файлу у пам'ять, що може спричинити проблеми з продуктивністю або пам'яттю при роботі з великими файлами.

11. Як відкрити текстовий файл для читання?
– Для відкриття текстового файлу для читання використовується `open('filename.txt', 'r')`.

12. Як записати дані у текстовий файл?
– Для запису даних у текстовий файл використовується `open('filename.txt', 'w')`, а потім метод `write()` для запису рядків.

13. Як зчитати текстовий файл построчно у список?
– Для зчитування текстового файлу построчно у список використовується метод `readlines()`, як у `lines = file.readlines()`.

14. Як відкрити бінарний файл для читання?
– Для відкриття бінарного файлу для читання використовується `open('filename.bin', 'rb')`.

15. Як записати дані у бінарний файл?
– Для запису даних у бінарний файл використовується `open('filename.bin', 'wb')`, а потім метод `write()` для запису байтів.

16. Як прочитати вміст бінарного файлу у одну змінну?
– Для зчитування вмісту бінарного файлу в одну змінну використовується метод `read()`, як у `content = file.read()`.

17. Як додати текст у кінець існуючого текстового файлу?
– Для додавання тексту у кінець існуючого текстового файлу використовується режим `'a'`, як у `with open('filename.txt', 'a') as file:`.

18. Як зчитати певну кількість байтів з бінарного файлу?
– Для зчитування певної кількості байтів з бінарного файлу використовується метод `read(size)`, як у `content = file.read(10)` для зчитування 10 байтів.

19. Як відкрити файл, щоб можна було і читати, і записувати текстові дані?

– Для відкриття файлу з можливістю читання і запису текстових даних використовується режим `'r+'`, як у `open('filename.txt', 'r+')`.

20. Як обробляти великі текстові файли, щоб уникнути завантаження всього файлу у пам'ять?

– Для обробки великих текстових файлів рекомендується використовувати построчне зчитування у циклі, як у:

```
with open('filename.txt', 'r') as file:
    for line in file:
        process(line)
```

Лекція №9. Обробка винятків

1. Огляд концепції винятків

Винятки (Exceptions) — це механізм в програмуванні, який дозволяє обробляти помилки та несподівані ситуації, що виникають під час виконання програми. Винятки дозволяють відокремити обробку помилок від основного коду програми, що робить код більш чистим і зрозумілим.

#Основні поняття:
- Виняток: Спеціальний об'єкт, який вказує на помилку або несподівану ситуацію.
- Перехоплення винятку: Процес реагування на виняток за допомогою конструкцій, які дозволяють обробити помилку без зупинки виконання програми.
- Базовий клас винятків: У Python всі винятки є підкласами базового класу `Exception`.

Приклад:

```
try:
    # Код, який може викликати виняток
    result = 10 / 0
except ZeroDivisionError:
    # Обробка винятку
    print("Помилка: ділення на нуль!")
```

В цьому прикладі, при спробі ділення на нуль виникає виняток `ZeroDivisionError`, який обробляється у блоці `except`.

2. Обробка винятків у Python за допомогою конструкції try-except

У Python обробка винятків реалізується за допомогою конструкції `try-except`. Конструкція `try` містить код, який може викликати виняток, а конструкція `except` — код для обробки винятку.

#Основи конструкції `try-except`:
- try: Вказує код, який може викликати виняток.
- except: Обробляє специфічний тип винятку, якщо він виникає.

Приклад:

```
try:
    num = int(input("Введіть число: "))
    result = 10 / num
except ValueError:
    print("Помилка: введено нечислове значення.")
except ZeroDivisionError:
    print("Помилка: ділення на нуль.")
else:
    print(f"Результат: {result}")
```

В цьому прикладі, якщо користувач введе нечислове значення, виникне `ValueError`, а якщо введе 0, то `ZeroDivisionError`. Якщо ж винятків не виникає, виконується блок `else`.

3. Використання конструкцій try-except-finally

Конструкція `try-except-finally` дозволяє виконати код в блоці `finally` незалежно від того, чи виник виняток у блоці `try`, чи ні. Це корисно для звільнення ресурсів, закриття файлів тощо.

#Основи конструкції `try-except-finally`:

- finally: Виконується завжди після виконання блоків `try` і `except`, незалежно від того, чи виник виняток.

Приклад:

```
def divide_numbers(a, b):
    try:
        result = a / b
        print(f"Результат: {result}")
    except ZeroDivisionError:
        print("Помилка: ділення на нуль.")
    finally:
        print("Кінець операції ділення.")

divide_numbers(10, 2)
divide_numbers(10, 0)
```

У цьому прикладі, блок `finally` виконується завжди, незалежно від того, чи виникає виняток у блоці `try`, чи ні.

Пояснення і практичні поради

1. Вибір типу винятку для обробки:
– Обробляйте конкретні типи винятків, щоб уникнути непередбачених помилок. Замість загального `except`, використовуйте специфічні винятки, як показано в прикладах.

2. Використання `else`:
– Блок `else` корисний для виконання коду, якщо жоден виняток не був піднятий. Це дозволяє краще структурувати код.

3. Ресурси та `finally`:

– Використовуйте блок `finally` для закриття файлів, звільнення ресурсів або виконання коду, який повинен бути виконаний завжди, наприклад, закриття бази даних.

4. Власні винятки:

– Розгляньте створення кастомних винятків для точнішої обробки помилок, специфічних для вашої програми.

Приклад кастомного винятку:

```
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

try:
    raise CustomError("Це кастомна помилка.")
except CustomError as e:
    print(f"Обробка кастомної помилки: {e}")
```

Ця лекція допоможе студентам краще зрозуміти концепцію винятків, їх обробку, та як використовувати конструкції для ефективного управління помилками в Python.

Питання для перевірки знань по темі лекції

10 питань для перевірки знань по темі "Огляд концепції винятків" в рамках лекції "Обробка винятків в Python", разом із правильними відповідями:

1. Що таке виняток у контексті програмування?

Відповідь: Виняток — це механізм для обробки помилок та несподіваних ситуацій, що виникають під час виконання програми. Винятки дозволяють відокремити обробку помилок від основного

коду, що допомагає зробити код більш структурованим і зручним для підтримки.

2. Яка роль базового класу `Exception` у Python?

Відповідь: Базовий клас `Exception` є суперкласом для всіх винятків у Python. Усі інші винятки є підкласами `Exception`, і він забезпечує базову функціональність для обробки помилок.

3. Як визначити, чи об'єкт є винятком у Python?

Відповідь: Об'єкт є винятком, якщо він є екземпляром класу, що успадковує від базового класу `Exception`. Це можна перевірити за допомогою функції `isinstance()`, наприклад: `isinstance(e, Exception)`.

4. Які основні типи винятків існують у Python?

Відповідь: Основні типи винятків у Python включають `ValueError`, `TypeError`, `IndexError`, `KeyError`, `FileNotFoundError`, `ZeroDivisionError`, `AttributeError`, та інші. Кожен тип винятку вказує на різний вид помилки.

5. Які є стандартні механізми для обробки винятків у Python?

Відповідь: Стандартні механізми для обробки винятків у Python включають конструкції `try`, `except`, `else`, і `finally`. Конструкція `try` містить код, що може викликати виняток, `except` обробляє винятки, `else` виконується, якщо винятків не було, а `finally` виконується завжди.

6. Що таке блок `finally`, і коли його слід використовувати?

Відповідь: Блок `finally` виконується завжди, незалежно від того, чи виник виняток у блоці `try`, чи ні. Його слід використовувати для виконання коду, який повинен бути виконаний завжди, наприклад, для закриття файлів або звільнення ресурсів.

7. Як можна вказати декілька типів винятків у конструкції `except`?

Відповідь: Для обробки декількох типів винятків в одному блоці `except` можна вказати кілька типів у кортежі. Наприклад:

```
try:
    # код
except (TypeError, ValueError) as e:
    print(f"Помилка: {e}")
```

8. Яка різниця між винятком і помилкою?

Відповідь: У загальному розумінні, виняток — це механізм обробки помилок у програмуванні, а помилка може бути будь-якою несподіваною ситуацією, що виникає під час виконання програми. Винятки спеціально обробляються у коді, а помилки — це загальні терміни, що описують проблему.

9. Як можна підняти власний виняток у Python, і для чого це може бути корисно?

Відповідь: Власний виняток можна підняти за допомогою ключового слова `raise` і створення кастомного класу винятку. Це корисно для створення специфічних помилок, що відповідають логіці програми. Наприклад:

```
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

raise CustomError("Це кастомна помилка.")
```

10. Які переваги має використання обробки винятків порівняно з простим перевіркою помилок за допомогою умов?

Відповідь: Обробка винятків дозволяє відокремити код для обробки помилок від основного коду, що робить програму більш читабельною і структурованою. Це також забезпечує централізовану обробку помилок, полегшуючи виявлення і усунення проблем. Умовні перевірки можуть перетворити код на заплутаний і менш гнучкий, особливо в складних програмах.

10 питань для перевірки знань по темі "Обробка винятків у Python за допомогою конструкції try-except" в рамках лекції "Обробка винятків в Python", разом із правильними відповідями:

1. Як працює конструкція `try-except` у Python?

Відповідь: Конструкція `try-except` у Python дозволяє перехоплювати і обробляти винятки, що виникають у блоці `try`. Код, що може викликати виняток, поміщається у блок `try`, а код для обробки винятків — у блок `except`. Якщо виняток виникає, управління передається в блок `except`.

2. Що трапиться, якщо у блоці `try` виникає виняток, а відповідного блоку `except` не знайдено?

Відповідь: Якщо у блоці `try` виникає виняток, а відповідного блоку `except` не знайдено, виняток буде піднятий далі по стеку викликів, і якщо він не буде оброблений десь ще, це призведе до завершення програми з повідомленням про помилку.

3. Як можна обробити кілька типів винятків у одному блоці `except`?

Відповідь: Для обробки кількох типів винятків у одному блоці `except` можна вказати кілька типів винятків у кортежі. Наприклад:

```
try:
    # код
except (TypeError, ValueError) as e:
    print(f"Помилка: {e}")
```

4. Яка мета використання ключового слова `as` у конструкції `except`?

Відповідь: Ключове слово `as` у конструкції `except` дозволяє зберігати об'єкт винятку в змінній, щоб отримати доступ до його атрибутів і повідомлень. Наприклад:

```
try:
    # код
except ValueError as e:
    print(f"Помилка: {e}")
```

У цьому прикладі змінна `e` містить інформацію про виняток.

5. Як можна ігнорувати певний тип винятків і продовжити виконання програми?

Відповідь: Щоб ігнорувати певний тип винятків і продовжити виконання програми, можна використовувати блок `except` без коду обробки або з порожнім тілом. Наприклад:

```
try:
   # код
except SomeException:
   pass
```

У цьому випадку виняток `SomeException` буде ігноруватися.

6. Яка різниця між блоком `except` і блоком `else` у конструкції `try-except`?

Відповідь: Блок `except` обробляє винятки, що виникають у блоці `try`, тоді як блок `else` виконується лише тоді, коли жодного винятку не виникло в блоці `try`. Блок `else` дозволяє виконати код, який повинен бути виконаний тільки при успішному виконанні блоку `try`.

7. Як можна виконати код, що повинен бути виконаний незалежно від того, чи виник виняток чи ні?

Відповідь: Для виконання коду, який повинен бути виконаний незалежно від того, чи виник виняток чи ні, використовують блок `finally`. Код у блоці `finally` завжди виконується після виконання блоків `try` і `except`.

8. Чи можна мати декілька блоків `except` для одного блоку `try`?

Відповідь: Так, можна мати декілька блоків `except` для одного блоку `try`. Це дозволяє обробляти різні типи винятків окремо. Наприклад:

```
try:
   # код
except ValueError:
   print("Обробка ValueError")
except TypeError:
   print("Обробка TypeError")
```

9. Що станеться, якщо виникне виняток у блоці `finally`?

Відповідь: Якщо виникне виняток у блоці `finally`, цей виняток замінить будь-який виняток, що був піднятий у блоці `try` або `except`. Виняток у блоці `finally` стане єдиним винятком, що обробляється, і код у блоках `try` і `except` не буде виконаний далі.

10. Як можна обробити виняток, що виникає в блоці `finally`?

Відповідь: Якщо потрібно обробити виняток, що виникає в блоці `finally`, можна обернути код у блоці `finally` в додатковий блок `try-except`. Це дозволяє обробити виняток, що виникає в `finally`, і уникнути неочікуваного завершення програми. Наприклад:

```
try:
   # код
finally:
   try:
      # код, що може викликати виняток
   except SomeException:
      print("Обробка винятку у блоці finally")
```

Ці питання допоможуть студентам глибше зрозуміти, як працює обробка винятків у Python за допомогою конструкції `try-except` та як правильно використовувати ці конструкції для обробки помилок у програмах.

10 питань для перевірки знань по темі "Використання конструкцій `try-except-finally`" в рамках лекції "Обробка винятків в Python", разом із правильними відповідями:

1. Що таке конструкція `try-except-finally` і яка її основна мета?

Відповідь: Конструкція `try-except-finally` є розширенням базової конструкції `try-except`, яка включає блок `finally`. Основна мета конструкції `try-except-finally` — забезпечити виконання певного коду (у блоці `finally`), незалежно від того, чи виник виняток у блоці `try`, чи ні. Це корисно для закриття файлів, звільнення ресурсів та інших завдань, що повинні бути виконані завжди.

2. Що відбудеться, якщо виникне виняток у блоці `finally`?

Відповідь: Якщо виникне виняток у блоці `finally`, цей виняток замінить будь-який виняток, що був піднятий у блоці `try` або `except`.

Це означає, що виняток у `finally` стане основним винятком, який буде переданий далі для обробки.

3. Чи може блок `finally` бути порожнім? Якщо так, то який його вплив на виконання програми?

Відповідь: Так, блок `finally` може бути порожнім. Якщо блок `finally` порожній, він просто виконається після блоку `try` і будь-якого блоку `except` без виконання будь-якого коду. Це все ще гарантує, що блок `finally` буде виконано, навіть якщо у блоці `try` виникає виняток.

4. Як конструкція `try-except-finally` може бути корисна для роботи з ресурсами, такими як файли або з'єднання з базою даних?

Відповідь: Конструкція `try-except-finally` корисна для забезпечення правильного закриття ресурсів, таких як файли або з'єднання з базою даних. Наприклад, незалежно від того, чи виник виняток під час обробки даних, блок `finally` може бути використаний для закриття файлів або з'єднань, що запобігає витоку ресурсів.

5. Який код виконається, якщо виникне виняток у блоці `try`, але блок `except` не обробляє цей виняток?

Відповідь: Якщо виникне виняток у блоці `try`, але блок `except` не обробляє цей виняток, то управління перейде до блоку `finally`, а потім виняток буде переданий далі по стеку викликів, що може призвести до завершення програми, якщо виняток не буде оброблений десь ще.

6. Як можна використовувати конструкцію `try-except-finally` для забезпечення належного закриття файлів у програмі?

Відповідь: Щоб забезпечити належне закриття файлів, можна використовувати конструкцію `try-except-finally`, де файл відкривається у блоці `try`, будь-які помилки обробляються у блоці `except`, а закриття файлу виконується в блоці `finally`. Наприклад:

```
try:
    file = open('example.txt', 'r')
    # обробка файлу
except IOError as e:
    print(f"Помилка вводу/виводу: {e}")
finally:
```

```
    file.close()
```

7. Чи може бути більше одного блоку `finally` у конструкції `try-except-finally`?

Відповідь: Ні, у конструкції `try-except-finally` може бути лише один блок `finally`. Кожен блок `try` може мати лише один блок `finally`, який виконується після блоків `try` і `except`.

8. Як ви можете обробити виняток у блоці `finally`, щоб запобігти його викиду?

Відповідь: Щоб обробити виняток у блоці `finally` і запобігти його викиду, можна обернути код у блоці `finally` у додатковий блок `try-except`. Це дозволяє обробити будь-який виняток, що виникає в `finally`, і уникнути його викиду. Наприклад:

```
try:
    # код
finally:
    try:
        # код, що може викликати виняток
    except SomeException as e:
        print(f"Обробка винятку у блоці finally: {e}")
```

9. Яка різниця між блоком `else` і блоком `finally` у конструкції `try-except`?

Відповідь: Блок `else` виконується лише тоді, коли жодного винятку не виникає в блоці `try`, тоді як блок `finally` виконується завжди, незалежно від того, чи виник виняток, чи ні. Блок `else` корисний для коду, що потрібно виконати, якщо винятків не було, а блок `finally` — для виконання коду, який повинен бути виконаний завжди.

10. Чи може конструкція `try-except-finally` бути використана для обробки винятків, що виникають у функціях, викликаних з блоку `try`?

Відповідь: Так, конструкція `try-except-finally` може бути використана для обробки винятків, що виникають у функціях, викликаних з блоку `try`. Якщо функція викликає виняток, цей виняток буде перехоплено блоком `except`, а блок `finally` буде виконано після обробки винятку,

забезпечуючи виконання необхідного коду для очистки або закриття ресурсів.

Ці питання дозволять студентам краще зрозуміти використання конструкції `try-except-finally` у Python і як ефективно використовувати ці конструкції для обробки винятків та забезпечення належного виконання коду.

Лекція №10. Робота з функціями та модулями

#1. Визначення та використання модулів у Python

Модулі – це файли з розширенням `.py`, які містять код Python (функції, класи, змінні), які можна повторно використовувати в інших програмах.

Приклад: Використання стандартного модуля `math`

```
import math

# Використання функції sqrt з модуля math
result = math.sqrt(16)
print(result)  # Виведе: 4.0

# Використання константи pi з модуля math
circumference = 2 * math.pi * 5
print(circumference)  # Виведе: 31.41592653589793
```

Ключові моменти:
- Модулі допомагають організовувати код.
- Можна імпортувати як стандартні, так і створені користувачем модулі.

#2. Створення та використання власних функцій

Функції дозволяють об'єднати кілька операторів в єдину логічну одиницю, яка може бути викликана в будь-якому місці програми.

Приклад: Створення функції для обчислення факторіалу числа

```
def factorial(n):
    """Обчислює факторіал числа n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

# Виклик функції
print(factorial(5))  # Виведе: 120
```

Ключові моменти:
- Функції дозволяють зменшити повторення коду.
- Функції можуть мати параметри та повертати значення.

#3. Імпортування функцій та модулів у Python

Імпортування дозволяє використовувати функції та змінні з інших модулів.

Приклад: Імпортування функції з іншого модуля

```
# Створення модуля my_module.py з функцією
# my_module.py
def greet(name):
    return f"Hello, {name}!"

# Імпортування та використання функції з модуля
# main.py
from my_module import greet
```

```
message = greet("Alice")
print(message)  # Виведе: Hello, Alice!
```

Приклад: Імпортування всього модуля

```
# Створення модуля another_module.py з кількома функціями
# another_module.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a – b

# Імпортування всього модуля
# main.py
import another_module

sum_result = another_module.add(10, 5)
print(sum_result)  # Виведе: 15

diff_result = another_module.subtract(10, 5)
print(diff_result)  # Виведе: 5
```

Ключові моменти:
- Імпортувати можна як окремі функції, так і цілі модулі.
- Використання `import` та `from ... import ...` забезпечує гнучкість у написанні коду.

Висновки

- Модулі дозволяють організувати код у більш зручну структуру.

- Функції полегшують повторне використання коду та забезпечують його логічну структуру.
- Імпортування модулів і функцій дозволяє використовувати їх в інших частинах програми або навіть у різних проектах.

Питання для перевірки знань по темі лекції

1. Що таке модуль у Python?
– Модуль у Python – це файл з розширенням `.py`, який містить код Python, включаючи функції, класи і змінні, які можуть бути повторно використані в інших програмах.

2. Як імпортувати стандартний модуль `math` у Python?
– Щоб імпортувати стандартний модуль `math`, використовують команду `import math`.

3. Яка команда дозволяє імпортувати лише функцію `sqrt` з модуля `math`?
– `from math import sqrt`

4. Як викликати функцію `sqrt` з модуля `math`, якщо модуль було імпортовано як `import math`?
– Викликати функцію можна за допомогою `math.sqrt()`.

5. Яка різниця між `import module_name` та `from module_name import function_name`?
– `import module_name` імпортує весь модуль, і до його функцій та змінних можна доступитися через синтаксис `module_name.function_name`. `from module_name import function_name` імпортує лише конкретну функцію або змінну з модуля, яку можна використовувати без префікса імені модуля.

6. Як створити власний модуль у Python?
– Власний модуль створюється шляхом написання коду Python у файл з розширенням `.py`. Цей файл можна імпортувати як модуль в інші програми.

7. Як імпортувати власний модуль `my_module.py`, що знаходиться у тому ж каталозі, що й основний скрипт?
– Імпортувати можна командою `import my_module`.

8. Як імпортувати всі функції з модуля `my_module`, не використовуючи префікс імені модуля?
– Використовується команда `from my_module import *`.

9. Що відбувається при імпортуванні модуля, якщо цей модуль містить виконуваний код (крім визначення функцій та класів)?
– При імпортуванні модуля виконується весь код цього модуля. Тобто всі команди, які не знаходяться в функціях або класах, будуть виконані.

10. Як уникнути виконання коду під час імпорту модуля?
– Виконуваний код слід помістити у блок `if __name__ == "__main__":`, що забезпечить його виконання лише при прямому запуску модуля як основного скрипта, а не при імпорті.

```
# my_module.py
def greet(name):
    return f"Hello, {name}!"

if __name__ == "__main__":
    print(greet("World"))
```

11. Що таке функція у Python?
– Функція у Python – це блок коду, який виконує певну задачу, і може бути викликаний кілька разів у програмі. Функції дозволяють організувати код, роблячи його більш читабельним та повторно використовуваним.

12. Як визначити функцію у Python?
– Функція визначається за допомогою ключового слова `def`, після якого йде ім'я функції, список параметрів у круглих дужках та двокрапка. Після цього записується тіло функції з відступом.

```
def my_function():
    # тіло функції
    pass
```

13. Як викликати функцію у Python?
– Функцію викликають за допомогою її імені, після якого йдуть круглі дужки. Якщо функція приймає аргументи, їх потрібно передати всередину дужок.

```
my_function()
```

14. Що таке параметри та аргументи функції у Python?
– Параметри – це змінні, зазначені у визначенні функції, які отримують значення під час виклику функції. Аргументи – це значення, які передаються функції при її виклику.

15. Як задати значення за замовчуванням для параметра функції?
– Значення за замовчуванням задається під час визначення функції за допомогою оператора `=`.

```
def greet(name="World"):
    print(f"Hello, {name}!")
```

16. Що таке ключові аргументи і як вони використовуються у Python?
– Ключові аргументи – це аргументи, передані функції за ім'ям, що дозволяє задавати значення параметрів у довільному порядку.

```
def greet(name, message):
    print(f"{message}, {name}!")

greet(message="Good morning", name="Alice")
```

17. Як повернути значення з функції у Python?
– Значення повертається з функції за допомогою ключового слова `return`.

```
def add(a, b):
    return a + b
```

18. Що станеться, якщо функція не містить ключового слова `return`?

– Якщо функція не містить ключового слова `return`, вона повертає значення `None`.

19. Що таке анонімні функції у Python і як їх створити?
– Анонімні функції у Python створюються за допомогою ключового слова `lambda`. Вони можуть мати кілька параметрів і одне вираз, який буде повернено.

```
add = lambda x, y: x + y
```

20. Як задати функцію з необмеженою кількістю аргументів?
– Для передачі необмеженої кількості позиційних аргументів використовують `*args`, а для ключових аргументів – `**kwargs`.

```
def print_args(*args):
    for arg in args:
        print(arg)

def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

21. Як імпортувати весь модуль у Python?
– Для імпортування всього модуля використовується команда `import module_name`.

```
import math
```

22. Як імпортувати лише певну функцію з модуля?
– Для імпортування певної функції використовується команда `from module_name import function_name`.

```
from math import sqrt
```

23. Як імпортувати модуль і надати йому інше ім'я?
– Для цього використовується ключове слово `as`.

```
import math as m
```

24. Як імпортувати кілька функцій з модуля?
– Для цього використовується команда `from module_name import function1, function2`.

```
from math import sqrt, ceil
```

25. Як імпортувати всі функції з модуля без використання імені модуля?
– Для цього використовується команда `from module_name import *`.

```
from math import *
```

26. Що таке відносне імпортування і коли воно використовується?
– Відносне імпортування використовується для імпорту модулів з поточного пакета або підпакетів. Воно використовується, коли структури каталогів складні і потрібно імпортувати модулі з тих самих або сусідніх каталогів.

```
from . import sibling_module
```

27. Як запобігти виконанню коду під час імпорту модуля?
– Код потрібно розмістити в блок `if __name__ == "__main__":`.

```
if __name__ == "__main__":
    # код, який не виконується під час імпорту
    print("This is the main module.")
```

28. Як дізнатися всі функції та змінні, доступні у модулі?
– Для цього можна використовувати функцію `dir()`.

```
import math
print(dir(math))
```

29. Що таке модуль `sys` і як його використовувати для додавання нових шляхів для пошуку модулів?
– Модуль `sys` містить змінні та функції, які впливають на роботу інтерпретатора Python. Для додавання нового шляху використовується список `sys.path`.

```
import sys
sys.path.append('/path/to/module')
```

30. Як повторно завантажити вже імпортований модуль?
– Для цього використовується функція `reload` з модуля `importlib`.

```
import importlib
import my_module
importlib.reload(my_module)
```

Лекція №11. Об'єктно-орієнтоване програмування (ООП)

#1. Вступ до концепції ООП

Основні поняття ООП:
- Клас: Шаблон або структура, яка визначає змінні та методи, що належать до об'єкта.
- Об'єкт: Екземпляр класу, який містить реальні значення змінних, визначених у класі.
- Інкапсуляція: Приховування деталей реалізації об'єкта та надання доступу до його стану тільки через методи.
- Наслідування: Механізм, який дозволяє створювати новий клас на основі існуючого класу.
- Поліморфізм: Здатність об'єктів різних класів реагувати на однакові методи по-різному.

#2. Класи та об'єкти у Python

Створення класу та об'єкта:

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        print(f"Car: {self.year} {self.make} {self.model}")

# Створення об'єкта
my_car = Car("Toyota", "Camry", 2020)
my_car.display_info()
```

Властивості та методи класу:
- Властивості (атрибути): Змінні, які зберігають стан об'єкта.
- Методи: Функції, які визначають поведінку об'єкта.

Приклад з інкапсуляцією:

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return True
        return False

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return True
        return False

    def get_balance(self):
        return self.__balance

# Створення об'єкта
account = BankAccount("John Doe", 1000)
account.deposit(500)
print(account.get_balance())  # 1500
```

#3. Наслідування та поліморфізм

Наслідування:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id

    def display_info(self):
        super().display_info()
        print(f"Student ID: {self.student_id}")

# Створення об'єкта
student = Student("Alice", 20, "S12345")
student.display_info()
```

Поліморфізм:

```
class Animal:
    def make_sound(self):
        raise NotImplementedError("Subclass must implement abstract method")
```

```
class Dog(Animal):
    def make_sound(self):
        return "Woof"

class Cat(Animal):
    def make_sound(self):
        return "Meow"

# Використання поліморфізму
def animal_sound(animal):
    print(animal.make_sound())

dog = Dog()
cat = Cat()
animal_sound(dog)  # Woof
animal_sound(cat)  # Meow
```

Підсумок

У цій лекції ми ознайомилися з основними концепціями ООП, створенням класів і об'єктів у Python, а також з механізмами наслідування та поліморфізму. Ці концепції є основоположними для розробки складних програмних систем і дозволяють створювати гнучкий та масштабований код.

Питання для перевірки знань по темі лекції

Питання для перевірки знань: "Вступ до концепції ООП у Python"

1. Що таке клас в об'єктно-орієнтованому програмуванні?
   – Відповідь: Клас – це шаблон або структура, яка визначає змінні та методи, що належать до об'єкта. Він описує загальні властивості та поведінку об'єктів.

2. Що таке об'єкт в об'єктно-орієнтованому програмуванні?
– Відповідь: Об'єкт – це екземпляр класу, який містить реальні значення змінних, визначених у класі, і може виконувати методи, визначені в класі.

3. Що таке інкапсуляція і як вона реалізується у Python?
– Відповідь: Інкапсуляція – це принцип ООП, який передбачає приховування деталей реалізації об'єкта та надання доступу до його стану тільки через методи. У Python інкапсуляція реалізується шляхом використання приватних атрибутів (починаються з двох підкреслювань, наприклад, `__balance`).

4. Що таке наслідування у ООП і як воно реалізується у Python?
– Відповідь: Наслідування – це механізм ООП, який дозволяє створювати новий клас на основі існуючого класу, успадковуючи його властивості та методи. У Python наслідування реалізується шляхом вказівки батьківського класу в дужках після імені дочірнього класу (наприклад, `class Student(Person)`).

5. Що таке поліморфізм і як він проявляється у Python?
– Відповідь: Поліморфізм – це здатність об'єктів різних класів реагувати на однакові методи по-різному. У Python поліморфізм досягається за допомогою методів, які мають однакові імена, але реалізовані в різних класах.

6. Наведіть приклад створення класу у Python з одним атрибутом і одним методом.
– Відповідь:

```
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        return "Woof!"
```

7. Що таке конструктор у класі і як його визначити у Python?
– Відповідь: Конструктор – це спеціальний метод класу, який автоматично викликається при створенні нового об'єкта. У Python конструктор визначається за допомогою методу `__init__`.

8. Як створити об'єкт класу і викликати його методи у Python?
– Відповідь:

```
class Cat:
    def __init__(self, name):
        self.name = name

    def meow(self):
        return "Meow!"

my_cat = Cat("Whiskers")
print(my_cat.meow())  # Виведе: Meow!
```

9. Що таке метод у класі і як він відрізняється від звичайної функції?
– Відповідь: Метод – це функція, яка визначена всередині класу і оперує об'єктами цього класу. Від звичайної функції метод відрізняється тим, що він приймає першим аргументом об'єкт (`self`), до якого він належить.

10. Як створити приватний атрибут у класі у Python?
– Відповідь: Приватний атрибут у класі створюється шляхом додавання двох підкреслювань на початку імені атрибута. Наприклад, `self.__balance`.

Питання для перевірки знань: "Класи та об'єкти у Python"

1. Що таке атрибути класу у Python і як їх визначити?
– Відповідь: Атрибути класу – це змінні, що належать класу. Їх можна визначити всередині методу `__init__`, використовуючи ключове слово `self`. Наприклад, `self.name = name`.

2. Як у Python визначити метод класу і чим він відрізняється від функції?
– Відповідь: Метод класу визначається всередині класу і приймає першим аргументом об'єкт (`self`). Він оперує даними, що належать цьому об'єкту. Наприклад:

```
class Person:
    def greet(self):
        print("Hello!")
```

3. Що таке конструктор у класі і як його використовувати у Python?
– Відповідь: Конструктор – це спеціальний метод `__init__`, який автоматично викликається при створенні нового об'єкта класу. Він використовується для ініціалізації атрибутів об'єкта.

4. Як створити об'єкт класу у Python?
– Відповідь: Об'єкт класу створюється шляхом виклику класу з необхідними аргументами (якщо такі є). Наприклад:

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

my_car = Car("Toyota", "Corolla")
```

5. Як отримати доступ до атрибутів об'єкта у Python?
– Відповідь: Доступ до атрибутів об'єкта здійснюється через нотацію крапки. Наприклад:

```
print(my_car.make)  # Toyota
```

6. Як визначити та використовувати метод класу у Python?
– Відповідь:

```
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        return "Woof!"

my_dog = Dog("Rex")
print(my_dog.bark())  # Woof!
```

7. Як змінити значення атрибута об'єкта у Python?
– Відповідь: Значення атрибута можна змінити, використовуючи нотацію крапки. Наприклад:

```
my_car.model = "Camry"
print(my_car.model)  # Camry
```

8. Що таке сам (`self`) у методах класу і для чого він використовується?
– Відповідь: `self` – це перший параметр будь-якого методу класу, який використовується для доступу до атрибутів та інших методів цього класу. Він вказує на конкретний об'єкт, з яким працює метод.

9. Як визначити статичний метод у класі у Python?
– Відповідь: Статичний метод визначається за допомогою декоратора `@staticmethod`. Він не приймає першим аргументом об'єкт `self` і може викликатися без створення об'єкта класу. Наприклад:

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b

print(Math.add(5, 3))  # 8
```

10. Як визначити метод класу, який працює з усім класом, а не з окремим об'єктом?
– Відповідь: Метод класу визначається за допомогою декоратора `@classmethod` і приймає першим аргументом клас (`cls`). Наприклад:

```
class Animal:
    species = "Canine"

    @classmethod
    def get_species(cls):
        return cls.species

print(Animal.get_species())  # Canine
```

Питання для перевірки знань: "Наслідування та поліморфізм у Python"

1. Що таке наслідування у Python і для чого воно використовується?
– Відповідь: Наслідування у Python – це механізм, який дозволяє створювати новий клас на основі існуючого класу, успадковуючи його

властивості та методи. Використовується для повторного використання коду та організації програмного забезпечення у ієрархії класів.

2. Як оголосити клас, який успадковує інший клас у Python?
– Відповідь: Щоб оголосити клас, який успадковує інший клас, необхідно вказати батьківський клас у дужках після імені дочірнього класу. Наприклад:

```
class Animal:
    pass

class Dog(Animal):
    pass
```

3. Як викликати метод батьківського класу у дочірньому класі?
– Відповідь: Метод батьківського класу можна викликати за допомогою функції `super()`. Наприклад:

```
class Animal:
    def sound(self):
        print("Some sound")

class Dog(Animal):
    def sound(self):
        super().sound()
        print("Woof")

dog = Dog()
dog.sound()
```

4. Що таке поліморфізм і як він реалізується у Python?
– Відповідь: Поліморфізм – це здатність об'єктів різних класів реагувати на однакові методи по-різному. У Python поліморфізм реалізується шляхом перевизначення методів у дочірніх класах.

5. Наведіть приклад перевизначення методу у дочірньому класі.
– Відповідь:

```
class Bird:
    def make_sound(self):
```

```
        return "Chirp"

    class Parrot(Bird):
        def make_sound(self):
            return "Squawk"

    parrot = Parrot()
    print(parrot.make_sound())  # Squawk
```

6. Що таке абстрактний клас і як він визначається у Python?
– Відповідь: Абстрактний клас – це клас, який не може бути інстанційований і призначений для визначення методів, які повинні бути реалізовані у дочірніх класах. У Python абстрактний клас визначається за допомогою модуля `abc`:

```
    from abc import ABC, abstractmethod

    class Animal(ABC):
        @abstractmethod
        def make_sound(self):
            pass
```

7. Як використовувати декоратор `@abstractmethod` у Python?
– Відповідь: Декоратор `@abstractmethod` використовується для позначення методів, які мають бути реалізовані у дочірніх класах. Наприклад:

```
    from abc import ABC, abstractmethod

    class Shape(ABC):
        @abstractmethod
        def area(self):
            pass
```

8. Що відбудеться, якщо не реалізувати абстрактний метод у дочірньому класі?
– Відповідь: Якщо не реалізувати абстрактний метод у дочірньому класі, спроба створити об'єкт цього класу викличе помилку `TypeError`,

оскільки абстрактні методи повинні бути реалізовані у всіх дочірніх класах.

9. Що таке множинне наслідування і як воно реалізується у Python?
– Відповідь: Множинне наслідування – це механізм, що дозволяє класу успадковувати властивості та методи від більше ніж одного батьківського класу. У Python це реалізується шляхом вказання кількох батьківських класів у дужках після імені дочірнього класу. Наприклад:

```
class Animal:
    pass

class Pet:
    pass

class Dog(Animal, Pet):
    pass
```

10. Що таке метод розв'язання порядку (MRO) і як його перевірити у Python?
– Відповідь: Метод розв'язання порядку (MRO) визначає порядок, у якому Python шукає методи у класах при множинному наслідуванні. MRO можна перевірити за допомогою атрибуту `__mro__` або функції `mro()`. Наприклад:

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

print(D.__mro__)  # (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

Лекція №12. Робота зі збірками

Збірки є основою для роботи з даними у Python. Основними типами збірок є списки, множини та кортежі. У цій лекції ми розглянемо, як працювати з цими збірками, їх основні операції та методи.

#Використання вбудованих збірок: списки, множини, кортежі

##Списки (Lists)
Список — це впорядкована колекція змінюваних елементів, які можуть бути різних типів.

Створення списків:

```
# Порожній список
empty_list = []

# Список з елементами
fruits = ['apple', 'banana', 'cherry']
```

Доступ до елементів списку:

```
print(fruits[0])  # apple
print(fruits[-1])  # cherry
```

Модифікація списку:

```
fruits[1] = 'blueberry'
print(fruits)  # ['apple', 'blueberry', 'cherry']
```

Додавання та видалення елементів:

```
# Додавання елемента
fruits.append('date')
print(fruits)  # ['apple', 'blueberry', 'cherry', 'date']

# Видалення елемента
fruits.remove('blueberry')
print(fruits)  # ['apple', 'cherry', 'date']
```

##Множини (Sets)
Множина — це невпорядкована колекція унікальних елементів.

Створення множин:

```
# Порожня множина
empty_set = set()

# Множина з елементами
colors = {'red', 'green', 'blue'}
```

Додавання та видалення елементів:

```
# Додавання елемента
colors.add('yellow')
print(colors)  # {'red', 'green', 'blue', 'yellow'}

# Видалення елемента
colors.remove('green')
print(colors)  # {'red', 'blue', 'yellow'}
```

Операції над множинами:

```
A = {1, 2, 3}
B = {3, 4, 5}

# Об'єднання
print(A | B)  # {1, 2, 3, 4, 5}

# Перетин
print(A & B)  # {3}

# Різниця
print(A – B)  # {1, 2}
```

##Кортежі (Tuples)
Кортеж — це впорядкована колекція незмінюваних елементів.

Створення кортежів:

```
# Порожній кортеж
empty_tuple = ()

# Кортеж з елементами
dimensions = (1920, 1080)
```

Доступ до елементів кортежу:

```
print(dimensions[0])  # 1920
print(dimensions[1])  # 1080
```

Кортежі є незмінюваними:

```
# Це викличе помилку
```

```
# dimensions[0] = 1280
```

#Операції та методи для роботи зі збірками

##Операції над списками
Додавання та злиття списків:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Додавання елементів
list1.append(4)
print(list1)  # [1, 2, 3, 4]

# Злиття списків
combined_list = list1 + list2
print(combined_list)  # [1, 2, 3, 4, 4, 5, 6]
```

Сортування списків:

```
numbers = [3, 1, 4, 1, 5, 9]
numbers.sort()
print(numbers)  # [1, 1, 3, 4, 5, 9]
```

##Операції над множинами
Перевірка наявності елементів:

```
fruits = {'apple', 'banana', 'cherry'}

print('banana' in fruits)  # True
print('date' in fruits)  # False
```

Інші операції над множинами:

```
A = {1, 2, 3}
B = {3, 4, 5}

# Симетрична різниця
print(A ^ B)  # {1, 2, 4, 5}
```

##Операції над кортежами
Злиття кортежів:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

combined_tuple = tuple1 + tuple2
print(combined_tuple)  # (1, 2, 3, 4, 5, 6)
```

Повторення кортежів:

```
tuple1 = (1, 2, 3)

repeated_tuple = tuple1 * 2
print(repeated_tuple)  # (1, 2, 3, 1, 2, 3)
```

#Висновок

У цій лекції ми розглянули три основні типи збірок у Python: списки, множини та кортежі. Кожен з цих типів має свої унікальні властивості та методи для роботи з ними. Знання цих збірок і вміння їх використовувати є важливим для ефективного програмування на Python.

Питання для перевірки знань по темі лекції

1. Що таке список у Python і як його створити?
– Список — це впорядкована колекція змінюваних елементів, які можуть бути різних типів. Для створення списку використовують квадратні дужки [].

```
my_list = [1, 2, 3, 'a', 'b']
```

2. Як додати елемент до списку у Python?
– Для додавання елемента до списку використовують метод `.append()`.

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # [1, 2, 3, 4]
```

3. Які існують способи доступу до елементів списку?
– Доступ до елементів списку здійснюється за допомогою індексів. Перший елемент має індекс 0, останній — -1.

```
my_list = [1, 2, 3]
print(my_list[0])  # 1
print(my_list[-1])  # 3
```

4. Що таке множина у Python і як її створити?
– Множина — це невпорядкована колекція унікальних елементів. Для створення множини використовують фігурні дужки {} або функцію `set()`.

```
my_set = {1, 2, 3}
empty_set = set()
```

5. Як видалити елемент з множини у Python?
– Для видалення елемента з множини використовують метод `.remove()` або `.discard()`.

```
my_set = {1, 2, 3}
my_set.remove(2)
print(my_set)  # {1, 3}
```

6. Що таке кортеж у Python і як його створити?
– Кортеж — це впорядкована колекція незмінюваних елементів. Для створення кортежу використовують круглі дужки ().

```
my_tuple = (1, 2, 3)
```

7. Як отримати доступ до елементів кортежу?
– Доступ до елементів кортежу здійснюється за допомогою індексів, так само як у списках.

```
my_tuple = (1, 2, 3)
print(my_tuple[0])  # 1
print(my_tuple[-1])  # 3
```

8. Які основні операції можна виконувати над множинами у Python?
– Основні операції: об'єднання (`|`), перетин (`&`), різниця (`-`), симетрична різниця (`^`).

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1 | set2)  # {1, 2, 3, 4, 5}
print(set1 & set2)  # {3}
print(set1 – set2)  # {1, 2}
print(set1 ^ set2)  # {1, 2, 4, 5}
```

9. Як змінити значення елемента у списку?
– Значення елемента у списку можна змінити за допомогою індексування.

```
my_list = [1, 2, 3]
my_list[1] = 'a'
print(my_list)  # [1, 'a', 3]
```

10. Чому не можна змінювати елементи кортежу після його створення?
– Кортежі є незмінюваними (immutable), тому їх елементи не можна змінювати після створення. Це властивість дозволяє використовувати кортежі як ключі в словниках або елементи множин.

```
my_tuple = (1, 2, 3)
# my_tuple[0] = 4  # Це викличе помилку
```

11. Як додати декілька елементів до списку за один раз?
– Використовуйте метод `.extend()`, щоб додати кілька елементів до списку.

```
my_list = [1, 2, 3]
my_list.extend([4, 5, 6])
print(my_list)  # [1, 2, 3, 4, 5, 6]
```

12. Як видалити елемент з певної позиції в списку?
– Використовуйте метод `.pop()`, щоб видалити елемент за індексом.

```
my_list = [1, 2, 3]
removed_element = my_list.pop(1)
print(my_list)  # [1, 3]
print(removed_element)  # 2
```

13. Як знайти індекс елемента в списку?
– Використовуйте метод `.index()`, щоб знайти індекс першого входження елемента.

```
my_list = ['a', 'b', 'c', 'b']
index = my_list.index('b')
print(index)  # 1
```

14. Як перевірити, чи є елемент в множині?
– Використовуйте оператор `in`, щоб перевірити наявність елемента в множині.

```
my_set = {1, 2, 3}
print(2 in my_set)  # True
print(4 in my_set)  # False
```

15. Як створити копію списку?
– Використовуйте метод `.copy()`, щоб створити поверхневу копію списку.

```
my_list = [1, 2, 3]
copy_list = my_list.copy()
print(copy_list)  # [1, 2, 3]
```

16. Як об'єднати два кортежі?
– Використовуйте оператор `+`, щоб об'єднати два кортежі.

```
tuple1 = (1, 2)
tuple2 = (3, 4)
combined_tuple = tuple1 + tuple2
print(combined_tuple)  # (1, 2, 3, 4)
```

17. Як видалити всі елементи зі списку?
– Використовуйте метод `.clear()`, щоб видалити всі елементи зі списку.

```
my_list = [1, 2, 3]
my_list.clear()
print(my_list)  # []
```

18. Як відсортувати список в зворотному порядку?
– Використовуйте метод `.sort(reverse=True)`, щоб відсортувати список в зворотному порядку.

```
my_list = [3, 1, 4, 1, 5]
my_list.sort(reverse=True)
print(my_list)  # [5, 4, 3, 1, 1]
```

19. Як обчислити кількість елементів у множині?
– Використовуйте функцію `len()`, щоб обчислити кількість елементів у множині.

```
my_set = {1, 2, 3, 4}
print(len(my_set))  # 4
```

20. Як повторити елементи кортежу кілька разів?
– Використовуйте оператор `*`, щоб повторити елементи кортежу кілька разів.

```
my_tuple = (1, 2, 3)
repeated_tuple = my_tuple * 2
print(repeated_tuple)  # (1, 2, 3, 1, 2, 3)
```

Лекція №13. Рекурсія

Рекурсія – це метод програмування, при якому функція викликає сама себе для розв'язання задачі. Рекурсивні алгоритми часто простіші та природніші для вирішення певних задач, але можуть вимагати більше ресурсів.

Приклад:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n – 1)
```

У цьому прикладі, `factorial` викликає саму себе для обчислення факторіалу числа `n`.

#Структура та властивості рекурсивних функцій

1. Базовий випадок (умова завершення):
   – Це частина функції, яка не викликає рекурсію і завершує процес.
   – Без базового випадку рекурсія триватиме нескінченно, викликаючи стекову переповнення.

Приклад:

```
def factorial(n):
    if n == 0:  # Базовий випадок
        return 1
    else:
        return n * factorial(n – 1)
```

2. Рекурсивний виклик :

– Це частина функції, яка викликає саму себе для обробки підзадачі.

– Кожен рекурсивний виклик повинен наближати задачу до базового випадку.

Приклад:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n – 1)  # Рекурсивний виклик
```

#Приклади рекурсивних алгоритмів

1. Обчислення факторіалу :

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n – 1)

print(factorial(5))  # Виведе 120
```

2. Числа Фібоначчі :

```
def fibonacci(n):
    if n <= 0:
```

```
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n – 1) + fibonacci(n – 2)

print(fibonacci(6))  # Виведе 8
```

3. Бінарний пошук :

```
def binary_search(arr, target, low, high):
    if low > high:
        return -1  # Елемент не знайдено
    mid = (low + high) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] > target:
        return binary_search(arr, target, low, mid – 1)
    else:
        return binary_search(arr, target, mid + 1, high)

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
target = 5
print(binary_search(arr, target, 0, len(arr) – 1))  # Виведе 4
```

4. Обхід дерева (приклад обходу дерева у глибину – DFS):

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

```
    def dfs(node):
        if node is not None:
            print(node.value)
            dfs(node.left)
            dfs(node.right)

    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)

    dfs(root)
```

#Переваги та недоліки рекурсії

Переваги:
- Простота реалізації для певних задач (наприклад, дерева, графи).
- Зрозумілість та читабельність коду.

Недоліки:
- Висока вартість у плані пам'яті через виклики функцій та збереження стану.
- Ризик стекового переповнення при глибоких рекурсіях.

Висновок
Рекурсія є потужним інструментом у програмуванні, який дозволяє вирішувати складні задачі простими і зрозумілими способами. Однак, необхідно використовувати її обережно, враховуючи можливі обмеження і недоліки.

Питання для перевірки знань по темі лекції

Питання для перевірки знань з теми "Огляд концепції рекурсії"

1. Що таке рекурсія в програмуванні?
– Рекурсія — це метод програмування, при якому функція викликає сама себе для вирішення задачі.

2. Чому важливий базовий випадок у рекурсивних функціях?
– Базовий випадок важливий для завершення рекурсії, оскільки без нього рекурсія триватиме нескінченно і призведе до переповнення стека.

3. Що станеться, якщо рекурсивна функція не має базового випадку?
– Якщо рекурсивна функція не має базового випадку, вона викликатиме сама себе нескінченно, що призведе до стекового переповнення.

4. Як рекурсивна функція наближається до базового випадку?
– Кожен рекурсивний виклик повинен зменшувати або змінювати параметри таким чином, щоб наблизити задачу до базового випадку.

5. Наведіть приклад простої рекурсивної функції, що обчислює факторіал числа.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n – 1)
```

6. Які два основні компоненти містить рекурсивна функція?
– Базовий випадок та рекурсивний виклик.

7. Чому рекурсія може бути більш природною для розв'язання певних задач?
– Деякі задачі, такі як обходи дерев або графів, мають природну рекурсивну структуру, що робить рекурсивні алгоритми простішими та зрозумілішими для реалізації.

8. Як можна уникнути переповнення стека при використанні рекурсії?

– Переконатися, що базовий випадок завжди досягається, і обмежити глибину рекурсії, якщо це можливо.

9. Чи всі задачі, що вирішуються рекурсивно, можна розв'язати ітеративно?

– Так, будь-яку задачу, що вирішується рекурсивно, можна розв'язати ітеративно, але ітеративний підхід може бути складнішим для реалізації в деяких випадках.

10. Чому рекурсивні функції можуть вимагати більше пам'яті, ніж ітеративні?

– Рекурсивні функції зберігають кожен виклик у стеку викликів, що може вимагати значних ресурсів пам'яті при глибокій рекурсії.

Питання для перевірки знань з теми "Структура та властивості рекурсивних функцій"

1. Що таке базовий випадок у рекурсивних функціях?

– Базовий випадок — це частина рекурсивної функції, яка завершує рекурсію, повертаючи конкретне значення без виклику самої себе.

2. Наведіть приклад базового випадку для функції, яка обчислює факторіал числа.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n – 1)
```

– Базовий випадок: `if n == 0: return 1`.

3. Що таке рекурсивний виклик у рекурсивній функції?

– Рекурсивний виклик — це частина функції, яка викликає саму себе для вирішення підзадачі.

4. Наведіть приклад рекурсивного виклику для функції, яка обчислює факторіал числа.

```
def factorial(n):
    if n == 0:
        return 1
```

```
    else:
        return n * factorial(n – 1)
```

– Рекурсивний виклик: `return n * factorial(n – 1)`.

5. Чому кожен рекурсивний виклик повинен наближати задачу до базового випадку?

– Кожен рекурсивний виклик повинен наближати задачу до базового випадку, щоб рекурсія завершилася і уникнути нескінченного циклу викликів, що може призвести до переповнення стека.

6. Як рекурсивні функції зберігають свій стан під час викликів?

– Рекурсивні функції зберігають свій стан за допомогою стека викликів, де кожен виклик зберігається з відповідними локальними змінними та параметрами.

7. Чому важливо переконатися, що рекурсивна функція не має нескінченного циклу викликів?

– Важливо уникнути нескінченного циклу викликів, оскільки це може призвести до переповнення стека, що зупинить виконання програми з помилкою.

8. Чим відрізняється хвостова рекурсія від звичайної рекурсії?

– У хвостовій рекурсії рекурсивний виклик є останньою операцією у функції, тоді як у звичайній рекурсії після рекурсивного виклику можуть бути ще інші операції.

9. Як можна оптимізувати рекурсивну функцію для обчислення чисел Фібоначчі?

– Використовувати мемоізацію або динамічне програмування для збереження результатів попередніх обчислень і уникнення повторних обчислень.

10. Які параметри рекурсивної функції потрібно змінювати при кожному рекурсивному виклику, щоб наближати задачу до базового випадку?

– Параметри, що використовуються для розбиття задачі на підзадачі, повинні змінюватися (наприклад, зменшуватися) при кожному рекурсивному виклику, щоб наближати задачу до базового випадку.

Питання для перевірки знань з теми "Приклади рекурсивних алгоритмів"

1. Як виглядає рекурсивна функція для обчислення чисел Фібоначчі?

```
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n – 1) + fibonacci(n – 2)
```

2. Наведіть приклад рекурсивної функції для знаходження найбільшого спільного дільника (НСД) двох чисел.

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

3. Як рекурсивно обчислити суму елементів списку?

```
def sum_list(lst):
    if not lst:
        return 0
    else:
        return lst[0] + sum_list(lst[1:])
```

4. Який алгоритм використовує рекурсію для сортування масиву елементів?

– Алгоритм швидкого сортування (QuickSort).

5. Як виглядає рекурсивна функція для швидкого сортування масиву?

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)
```

6. Наведіть приклад рекурсивної функції для обходу бінарного дерева у глибину (DFS).

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def dfs(node):
    if node is not None:
        print(node.value)
        dfs(node.left)
        dfs(node.right)
```

7. Як рекурсивно перевірити, чи є слово паліндромом?

```
def is_palindrome(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    return is_palindrome(s[1:-1])
```

8. Як можна рекурсивно знайти мінімальний елемент у списку?

```
def find_min(lst):
    if len(lst) == 1:
        return lst[0]
    else:
        min_rest = find_min(lst[1:])
        return min(lst[0], min_rest)
```

9. Чим рекурсивний бінарний пошук відрізняється від ітеративного?

– Рекурсивний бінарний пошук використовує рекурсивні виклики для поділу масиву на підмасиви, тоді як ітеративний використовує цикли для тієї ж мети.

10. Як виглядає рекурсивна функція для обчислення степеня числа?

```
def power(base, exp):
```

```
if exp == 0:
   return 1
else:
   return base * power(base, exp – 1)
```

Лекція №14. Сортування та пошук

Сортування та пошук — це основні операції в багатьох алгоритмах і структурах даних. Ефективні методи сортування та пошуку дозволяють значно прискорити роботу програм, особливо при обробці великих обсягів даних. У цій лекції ми розглянемо кілька популярних алгоритмів сортування та пошуку.

Сортування

#Сортування вставками (Insertion Sort)

Опис алгоритму:
Сортування вставками працює за принципом поступового нарощування відсортованого підмасиву. На кожній ітерації алгоритм бере один елемент з несортованої частини масиву і вставляє його у відповідне місце у відсортованій частині.

Псевдокод:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i – 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Приклад:

```
arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
print("Sorted array is:", arr)
# Вихід: [5, 6, 11, 12, 13]
```

#Сортування обміном (Bubble Sort)

Опис алгоритму:

Сортування обміном працює, багаторазово проходячи по списку, порівнюючи сусідні елементи та обмінюючи їх, якщо вони знаходяться у неправильному порядку. Цей процес повторюється до тих пір, поки список не буде відсортований.

Псевдокод:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Приклад:

```
arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print("Sorted array is:", arr)
# Вихід: [11, 12, 22, 25, 34, 64, 90]
```

#Швидке сортування (Quick Sort)

Опис алгоритму:

Швидке сортування використовує підхід розділяй і володарюй. Вибирається опорний елемент, після чого масив розділяється на два підмасиви: елементи, менші за опорний, і елементи, більші за опорний. Далі ці підмасиви сортуються рекурсивно.

Псевдокод:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2]
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]
        return quick_sort(left) + middle + quick_sort(right)
```

Приклад:

```
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = quick_sort(arr)
print("Sorted array is:", sorted_arr)
# Вихід: [1, 1, 2, 3, 6, 8, 10]
```

Пошук

#Послідовний пошук (Sequential Search)

Опис алгоритму:

Послідовний пошук проходить по всіх елементах масиву до тих пір, поки не буде знайдений потрібний елемент або не будуть перевірені всі елементи.

Псевдокод:

```
def sequential_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
```

Приклад:

```
arr = [2, 3, 4, 10, 40]
x = 10
result = sequential_search(arr, x)
if result != -1:
    print("Element found at index", result)
else:
    print("Element not found")
# Вихід: Element found at index 3
```

#Бінарний пошук (Binary Search)

Опис алгоритму:
Бінарний пошук працює тільки з відсортованими масивами. Алгоритм порівнює середній елемент масиву з шуканим значенням і рекурсивно продовжує пошук у лівій або правій половині масиву в залежності від результату порівняння.

Псевдокод:

```
def binary_search(arr, x):
    low = 0
    high = len(arr) – 1
    mid = 0

    while low <= high:
        mid = (high + low) // 2
        if arr[mid] < x:
            low = mid + 1
        elif arr[mid] > x:
            high = mid – 1
        else:
            return mid
    return -1
```

Приклад:

```
arr = [2, 3, 4, 10, 40]
x = 10
result = binary_search(arr, x)
if result != -1:
    print("Element found at index", result)
else:
    print("Element not found")
# Вихід: Element found at index 3
```

Висновок

Сортування та пошук є ключовими навичками для ефективної обробки даних у програмуванні. Знання різних алгоритмів сортування та

пошуку дозволяє вибирати оптимальний метод для кожного конкретного завдання.

Питання для перевірки знань по темі лекції

1. Опишіть основні етапи роботи алгоритму сортування вставками.

Відповідь:
– Починаючи з другого елемента масиву, кожен елемент порівнюється з попередніми елементами в масиві.
– Якщо елемент менший за попередній, він переміщується назад до тих пір, поки не буде знайдено місце, де він більший за попередній елемент або дійде до початку масиву.
– Процес повторюється для всіх елементів масиву.

2. Яка основна ідея сортування обміном і як воно працює?

Відповідь:
– Основна ідея сортування обміном (bubble sort) полягає в тому, щоб багаторазово проходити по списку, порівнюючи сусідні елементи та обмінюючи їх, якщо вони знаходяться у неправильному порядку.
– Цей процес повторюється до тих пір, поки не буде зроблено жодного обміну під час проходу через масив.

3. Що таке опорний елемент у швидкому сортуванні і як він використовується?

Відповідь:
– Опорний елемент (pivot) вибирається з масиву для розділення масиву на два підмасиви.
– Один підмасив містить елементи, менші за опорний, інший — елементи, більші за опорний.
– Після розділення ці підмасиви сортуються рекурсивно за тим же принципом.

4. Який час виконання алгоритму сортування вставками в середньому випадку?

Відповідь:
– Час виконання алгоритму сортування вставками в середньому випадку — O(n^2).

5. Поясніть різницю між найгіршим і найкращим часом виконання для сортування обміном.

Відповідь:
– Найгірший час виконання сортування обміном — O(n^2), що відбувається, коли масив відсортований у зворотному порядку.
– Найкращий час виконання — O(n), коли масив вже відсортований і не потрібно жодного обміну.

6. Який середній час виконання швидкого сортування і від чого він залежить?

Відповідь:
– Середній час виконання швидкого сортування — O(n log n).
– Він залежить від того, як добре вибраний опорний елемент. Якщо опорний елемент завжди розділяє масив приблизно на дві рівні частини, час виконання буде мінімальним.

7. Опишіть процес розділення масиву в алгоритмі швидкого сортування.

Відповідь:
– Вибирається опорний елемент.
– Всі елементи, менші за опорний, переміщуються в ліву частину масиву, а всі елементи, більші за опорний, переміщуються в праву частину.
– Опорний елемент займає своє кінцеве місце в масиві.
– Процес повторюється рекурсивно для підмасивів зліва і справа від опорного елемента.

8. Назвіть основну перевагу швидкого сортування над іншими алгоритмами сортування.

Відповідь:
– Основна перевага швидкого сортування — його середній час виконання O(n log n), що робить його значно швидшим за алгоритми з часом виконання O(n^2) для великих масивів.

9. Як можна поліпшити продуктивність швидкого сортування у випадку з вибором невдалого опорного елемента?

Відповідь:

– Використання методів вибору опорного елемента, таких як медіана з трьох (median-of-three) або випадковий вибір (randomized quicksort), щоб зменшити ймовірність вибору найгіршого опорного елемента.

10. Які типи масивів найкраще підходять для кожного з алгоритмів сортування (вставками, обміном, швидким)?

Відповідь:
– Сортування вставками найкраще підходить для майже відсортованих масивів або невеликих масивів.
– Сортування обміном не має жодних конкретних переваг для будь-якого типу масиву через його низьку ефективність.
– Швидке сортування найкраще підходить для великих масивів, де очікується різнорідний розподіл даних.

11. Опишіть основні етапи роботи алгоритму послідовного пошуку.

Відповідь:
– Алгоритм послідовно перевіряє кожен елемент масиву, починаючи з першого.
– Порівнює кожен елемент з шуканим значенням.
– Якщо знаходить елемент, що дорівнює шуканому значенню, повертає його індекс.
– Якщо не знаходить жодного збігу до кінця масиву, повертає -1.

12. Який час виконання алгоритму послідовного пошуку в найгіршому випадку?

Відповідь:
– Час виконання в найгіршому випадку — O(n), де n — кількість елементів у масиві.

13. Опишіть основні етапи роботи алгоритму бінарного пошуку.

Відповідь:
– Масив має бути відсортованим.
– Встановлюються початкові межі низького та високого індексів.
– Поки межі не перетнуться, визначається середній індекс і порівнюється середній елемент з шуканим значенням.
– Якщо середній елемент дорівнює шуканому значенню, алгоритм повертає його індекс.

– Якщо середній елемент менший за шукане значення, низький індекс зміщується праворуч.
– Якщо середній елемент більший за шукане значення, високий індекс зміщується ліворуч.
– Якщо межі перетнулись і значення не знайдено, алгоритм повертає -1.

14. Який час виконання алгоритму бінарного пошуку в найгіршому випадку?

Відповідь:
– Час виконання в найгіршому випадку — O(log n), де n — кількість елементів у масиві.

15. Чому для бінарного пошуку необхідно, щоб масив був відсортованим?

Відповідь:
– Бінарний пошук працює за принципом поділу масиву на дві частини. Без відсортованого масиву не можна гарантовано визначити, в якій половині знаходиться шуканий елемент, оскільки порядок елементів буде невідомий.

16. Які переваги та недоліки послідовного пошуку порівняно з бінарним пошуком?

Відповідь:
– Переваги: простий у реалізації, не потребує відсортованого масиву, працює для будь-якого типу даних.
– Недоліки: повільний для великих масивів, час виконання O(n).
– Переваги бінарного пошуку: швидший для великих відсортованих масивів, час виконання O(log n).
– Недоліки бінарного пошуку: потребує відсортованого масиву, складніший у реалізації.

17. Як модифікувати алгоритм бінарного пошуку для пошуку першого або останнього входження елемента у відсортованому масиві?

Відповідь:
– Пошук першого входження: після знаходження елемента продовжити пошук ліворуч до тих пір, поки не буде знайдено найлівіше входження.

– Пошук останнього входження: після знаходження елемента продовжити пошук праворуч до тих пір, поки не буде знайдено найправіше входження.

18. Які дані можна використовувати для послідовного та бінарного пошуку?

Відповідь:
– Послідовний пошук: будь-які дані (числові, рядкові, об'єкти тощо).
– Бінарний пошук: будь-які відсортовані дані, які можна порівнювати (числові, рядкові, об'єкти з визначеним порядком).

19. Як можна покращити ефективність послідовного пошуку?

Відповідь:
– Використання індексації або хешування для прискорення пошуку в великих масивах.
– Паралельне виконання послідовного пошуку на великих наборах даних.

20. Які помилки можуть виникнути при реалізації алгоритму бінарного пошуку?

Відповідь:
– Використання не відсортованого масиву.
– Неправильне обчислення середнього індексу, що може призвести до нескінченного циклу.
– Невірне оновлення меж (низького або високого індексу) після порівняння середнього елемента з шуканим значенням.

Лекція №15. Паралельне програмування

План
1. Огляд концепції паралельного програмування
   – Визначення паралельного програмування.
   – Переваги та виклики паралельного програмування.
   – Паралельні обчислення vs. багатозадачність.
   – Концепції потоків та процесів.

2. Модуль `multiprocessing` у Python
   – Огляд модуля `multiprocessing`.
   – Основні компоненти модуля: Process, Pool, Queue, Pipe, Lock.
   – Створення та керування процесами.
   – Синхронізація процесів та обмін даними між ними.

3. Застосування паралельного програмування для вирішення задач
   – Вступ до прикладних задач.
   – Використання `multiprocessing` для обробки великих обсягів даних.
   – Приклад реалізації паралельного алгоритму.
   – Оптимізація програм з використанням паралельних обчислень.

1. Огляд концепції паралельного програмування

#Визначення паралельного програмування
Паралельне програмування — це метод програмування, при якому кілька обчислювальних завдань виконуються одночасно. Це дозволяє ефективніше використовувати ресурси системи, такі як процесорні ядра, і значно прискорює обчислювальні процеси.

#Переваги та виклики паралельного програмування
Переваги:
- Збільшення швидкості виконання програм.
- Ефективне використання багатоядерних процесорів.
- Можливість обробки великих обсягів даних у реальному часі.

Виклики:
- Складність синхронізації між процесами.
- Ризик виникнення стану гонки (race conditions).
- Збільшення складності налагодження та тестування програм.

#Паралельні обчислення vs. багатозадачність
- Паралельні обчислення : одночасне виконання кількох завдань для пришвидшення обчислень.
- Багатозадачність : здатність системи виконувати кілька завдань, переключаючи контекст між ними.

#Концепції потоків та процесів
- Потоки : легкі одиниці виконання всередині процесу, які розділяють спільну пам'ять.
- Процеси : незалежні програми з власним адресним простором пам'яті.

2. Модуль `multiprocessing` у Python

#Огляд модуля `multiprocessing`
Модуль `multiprocessing` надає можливості для створення паралельних програм у Python, дозволяючи створювати та керувати процесами.

#Основні компоненти модуля
- Process : базовий клас для створення процесів.
- Pool : клас для управління групою робітничих процесів.
- Queue : черга для обміну даними між процесами.
- Pipe : двосторонній канал для зв'язку між процесами.
- Lock : механізм для синхронізації доступу до ресурсів.

#Створення та керування процесами

Приклад створення процесу :

```
from multiprocessing import Process

def print_square(num):
    print(f'Square: {num * num}')

if __name__ == "__main__":
    p1 = Process(target=print_square, args=(10,))
    p1.start()
    p1.join()
```

#Синхронізація процесів та обмін даними між ними

Приклад використання Queue для обміну даними :

```
from multiprocessing import Process, Queue

def producer(queue):
    for i in range(5):
        queue.put(i * i)

def consumer(queue):
    while not queue.empty():
        print(queue.get())

if __name__ == "__main__":
    q = Queue()
    p1 = Process(target=producer, args=(q,))
    p2 = Process(target=consumer, args=(q,))

    p1.start()
    p1.join()

    p2.start()
    p2.join()
```

3. Застосування паралельного програмування для вирішення задач

#Вступ до прикладних задач
Паралельне програмування використовується у різних областях: обробка великих обсягів даних, машинне навчання, наукові обчислення, симуляції.

#Використання `multiprocessing` для обробки великих обсягів даних

Приклад обробки великого масиву даних :

```
from multiprocessing import Pool

def square_number(num):
    return num * num

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5]
```

```
    with Pool() as pool:
        result = pool.map(square_number, numbers)
    print(result)
```

#Приклад реалізації паралельного алгоритму
Паралельний підхід до сортування :

```
from multiprocessing import Pool

def merge_sort_parallel(data):
    if len(data) <= 1:
        return data
    mid = len(data) // 2
    with Pool() as pool:
        left, right = pool.map(merge_sort_parallel, [data[:mid], data[mid:]])
    return merge(left, right)

def merge(left, right):
    result = []
    while left and right:
        if left[0] < right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    result.extend(left or right)
    return result

if __name__ == "__main__":
    data = [38, 27, 43, 3, 9, 82, 10]
    sorted_data = merge_sort_parallel(data)
    print(sorted_data)
```

#Оптимізація програм з використанням паралельних обчислень
Оптимізація програм включає розподіл завдань на дрібніші підзадачі, використання ефективних алгоритмів та зменшення накладних витрат на синхронізацію.

Висновок
Паралельне програмування в Python з використанням модуля `multiprocessing` дозволяє значно прискорити виконання програм,

обробляючи великі обсяги даних та виконуючи складні обчислення. Важливо розуміти основні принципи паралельного програмування та правильно застосовувати їх у своїх задачах.

Питання для перевірки знань по темі лекції

Питання для перевірки знань

1. Що таке паралельне програмування?
   – Відповідь : Паралельне програмування — це метод програмування, при якому кілька обчислювальних завдань виконуються одночасно, що дозволяє ефективніше використовувати ресурси системи, такі як процесорні ядра, і значно прискорює обчислювальні процеси.

2. Які основні переваги паралельного програмування?
   – Відповідь : Основні переваги паралельного програмування включають збільшення швидкості виконання програм, ефективне використання багатоядерних процесорів, та можливість обробки великих обсягів даних у реальному часі.

3. Які виклики виникають при паралельному програмуванні?
   – Відповідь : Виклики включають складність синхронізації між процесами, ризик виникнення стану гонки (race conditions), та збільшення складності налагодження та тестування програм.

4. Чим відрізняється паралельне обчислення від багатозадачності?
   – Відповідь : Паралельні обчислення передбачають одночасне виконання кількох завдань для пришвидшення обчислень, тоді як багатозадачність передбачає здатність системи виконувати кілька завдань, переключаючи контекст між ними.

5. Що таке потоки в контексті паралельного програмування?

– Відповідь : Потоки — це легкі одиниці виконання всередині процесу, які розділяють спільну пам'ять.

6. Що таке процеси в контексті паралельного програмування?
– Відповідь : Процеси — це незалежні програми з власним адресним простором пам'яті.

7. Наведіть приклад практичної задачі, де паралельне програмування є необхідним.
– Відповідь : Паралельне програмування є необхідним при обробці великих обсягів даних, наприклад, аналіз великих наборів даних (Big Data), де обробка кожного запису може бути виконана незалежно.

8. Які основні проблеми можуть виникати при синхронізації процесів?
– Відповідь : Основні проблеми включають виникнення стану гонки, блокування процесів (deadlock), неправильний порядок виконання (race condition), та накладні витрати на синхронізацію.

9. Що таке стан гонки (race condition) і як його уникнути?
– Відповідь : Стан гонки виникає, коли два або більше процесів/потоків мають доступ до спільного ресурсу і намагаються змінити його одночасно. Його можна уникнути за допомогою механізмів синхронізації, таких як блокування (locks), семафори (semaphores), та монітори (monitors).

10. Як паралельне програмування впливає на продуктивність програм?
– Відповідь : Паралельне програмування може значно підвищити продуктивність програм, дозволяючи виконувати кілька обчислювальних завдань одночасно, що скорочує загальний час виконання завдання, особливо на багатоядерних процесорах.

11. Що таке модуль `multiprocessing` у Python і для чого він використовується?

– Відповідь : Модуль `multiprocessing` у Python використовується для створення паралельних програм, дозволяючи створювати та керувати процесами для виконання завдань одночасно на декількох процесорних ядрах.

12. Які основні компоненти модуля `multiprocessing`?

– Відповідь : Основні компоненти модуля `multiprocessing` включають Process, Pool, Queue, Pipe, Lock.

13. Як створити новий процес за допомогою модуля `multiprocessing`?

– Відповідь : Новий процес можна створити за допомогою класу `Process`. Наприклад:

```
from multiprocessing import Process

def worker():
    print('Worker function')

if __name__ == "__main__":
    p = Process(target=worker)
    p.start()
    p.join()
```

14. Що таке Pool і для чого він використовується?

– Відповідь : Pool — це клас, який використовується для управління пулом робочих процесів, дозволяючи легко виконувати завдання паралельно, розподіляючи їх між процесами в пулі.

15. Як використовувати клас Queue для обміну даними між процесами?

– Відповідь : Клас `Queue` використовується для безпечного обміну даними між процесами. Приклад:

```
from multiprocessing import Process, Queue

def producer(queue):
    queue.put('Data')

def consumer(queue):
    data = queue.get()
    print(data)

if __name__ == "__main__":
    q = Queue()
    p1 = Process(target=producer, args=(q,))
    p2 = Process(target=consumer, args=(q,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

16. Що таке Pipe і як його використовувати для зв'язку між процесами?

– Відповідь : Pipe — це двосторонній канал для зв'язку між процесами. Його можна використовувати так:

```
from multiprocessing import Process, Pipe

def sender(conn):
```

```
    conn.send('Hello')
    conn.close()

def receiver(conn):
    print(conn.recv())
    conn.close()

if __name__ == "__main__":
    parent_conn, child_conn = Pipe()
    p1 = Process(target=sender, args=(child_conn,))
    p2 = Process(target=receiver, args=(parent_conn,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

17. Для чого використовується клас Lock і як його застосовувати?

– Відповідь : Клас Lock використовується для синхронізації доступу до спільних ресурсів, щоб уникнути стану гонки. Приклад:

```
from multiprocessing import Process, Lock

def worker(lock, num):
    with lock:
        print(f'Worker {num}')

if __name__ == "__main__":
    lock = Lock()
    processes = [Process(target=worker, args=(lock, i)) for i in range(5)]
    for p in processes:
        p.start()
    for p in processes:
```

```
    p.join()
```

18. Як за допомогою модуля `multiprocessing` можна розподілити завдання між кількома процесами?

– Відповідь : Завдання можна розподілити між кількома процесами за допомогою класу `Pool` і методу `map`. Приклад:

```
from multiprocessing import Pool

def square(x):
    return x * x

if __name__ == "__main__":
    with Pool(4) as p:
        results = p.map(square, [1, 2, 3, 4])
    print(results)
```

19. Як зупинити процес, створений за допомогою модуля `multiprocessing`?

– Відповідь : Процес можна зупинити за допомогою методу `terminate()`. Приклад:

```
from multiprocessing import Process
import time

def worker():
    while True:
        print('Working...')
        time.sleep(1)

if __name__ == "__main__":
    p = Process(target=worker)
```

```
    p.start()
    time.sleep(5)
    p.terminate()
```

20. Як перевірити, чи процес ще працює, використовуючи модуль `multiprocessing`?

– Відповідь : Можна використовувати атрибут `is_alive()`. Приклад:

```
from multiprocessing import Process
import time

def worker():
    time.sleep(2)

if __name__ == "__main__":
    p = Process(target=worker)
    p.start()
    print(p.is_alive())
    p.join()
    print(p.is_alive())
```

21. Як паралельне програмування може допомогти в обробці великих обсягів даних?

– Відповідь : Паралельне програмування дозволяє розподілити обробку великих обсягів даних між кількома процесами, що працюють одночасно, тим самим значно зменшуючи час обробки.

22. Які прикладні задачі можуть бути вирішені за допомогою паралельного програмування?

– Відповідь : Прикладні задачі включають машинне навчання, обробку великих даних (Big Data), наукові обчислення, рендеринг графіки, обробку зображень та відео, і симуляції.

23. Як використання класу Pool допомагає при вирішенні задач?

– Відповідь : Клас `Pool` дозволяє легко розподіляти завдання між групою робочих процесів, що значно спрощує паралельне виконання завдань та підвищує ефективність обчислень.

24. Наведіть приклад задачі, де можна використати Pool для паралельної обробки.

– Відповідь : Обробка великого масиву чисел для обчислення їх квадратів може бути ефективно реалізована за допомогою `Pool`:

```
from multiprocessing import Pool

def square(number):
    return number * number

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5]
    with Pool() as pool:
        results = pool.map(square, numbers)
    print(results)
```

25. Як синхронізація процесів може вплинути на продуктивність паралельних програм?

– Відповідь : Неправильна синхронізація процесів може призвести до стану гонки, блокування та інших проблем, які негативно впливають на продуктивність і надійність програм.

26. Як можна уникнути стану гонки при паралельному програмуванні?

– Відповідь : Стану гонки можна уникнути, використовуючи механізми синхронізації, такі як блокування (Locks), семафори (Semaphores), монітори (Monitors) та інші засоби контролю доступу до спільних ресурсів.

27. Опишіть, як обробляти дані з використанням Queue для паралельного програмування.

– Відповідь : Queue дозволяє безпечно передавати дані між процесами. Наприклад, один процес може поміщати дані в чергу, а інший — отримувати і обробляти їх:

```
from multiprocessing import Process, Queue

def producer(queue):
    for i in range(5):
        queue.put(i * i)

def consumer(queue):
    while not queue.empty():
        print(queue.get())

if __name__ == "__main__":
    q = Queue()
    p1 = Process(target=producer, args=(q,))
    p2 = Process(target=consumer, args=(q,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()
```

28. Чому важливо використовувати паралельне програмування при обробці зображень?

– Відповідь : Обробка зображень зазвичай включає обчислювально інтенсивні операції, такі як фільтрація, перетворення, розпізнавання об'єктів тощо. Паралельне програмування дозволяє значно прискорити ці операції, розподіляючи їх між кількома процесорами.

29. Яким чином можна використовувати паралельне програмування для обробки великих лог-файлів?

– Відповідь : Лог-файли можна розділити на частини та обробляти кожну частину паралельно за допомогою окремих процесів, що дозволяє швидше аналізувати велику кількість записів.

30. Які стратегії можна застосувати для оптимізації паралельних обчислень у Python?

– Відповідь : Стратегії включають:

– Розподіл завдань на незалежні підзадачі.

– Використання ефективних алгоритмів.

– Мінімізація накладних витрат на синхронізацію.

– Використання відповідних структур даних для обміну інформацією між процесами.

– Проведення профілювання і налаштування програм для виявлення і усунення вузьких місць.

Лекція №16. Проекти та практичні завдання

План лекції:

1. Вступ
2. Обговорення проектів та практичних завдань
   – Проекти з обробки даних
   – Проекти з веб-розробки
   – Проекти з автоматизації
   – Проекти з машинного навчання
3. Приклади проектів та практичних завдань
   – Детальні приклади з поясненнями
4. Рекомендації щодо подальшого самостійного навчання та розвитку у програмуванні на мові Python
   – Вибір ресурсів для навчання
   – Поради щодо практики програмування
   – Залучення до спільнот та участь у конкурсах
5. Висновок

1. Вступ

Ця лекція спрямована на обговорення проектів та практичних завдань, які можна виконувати, використовуючи набуті знання з Python. Також будуть надані рекомендації щодо подальшого самостійного навчання та розвитку у програмуванні.

2. Обговорення проектів та практичних завдань

#Проекти з обробки даних

Обробка даних є однією з найпоширеніших сфер застосування Python. Знання Pandas, NumPy та інших бібліотек дозволяє вирішувати такі завдання:
- Аналіз даних: використання Pandas для зчитування, обробки та аналізу великих наборів даних.
- Візуалізація даних: створення графіків та діаграм за допомогою бібліотек Matplotlib та Seaborn.

#Проекти з веб-розробки

Python широко використовується у веб-розробці завдяки фреймворкам, таким як Django та Flask:
- Створення веб-сайтів та веб-додатків: від простих сайтів до складних додатків.
- Розробка API: створення RESTful API для взаємодії з іншими додатками.

#Проекти з автоматизації

Автоматизація рутинних завдань є однією з корисних сфер застосування Python:
- Автоматизація роботи з файлами та папками.
- Автоматизація збору даних з веб-сайтів (web scraping).

#Проекти з машинного навчання

Машинне навчання стає все більш популярним завдяки бібліотекам, таким як Scikit-Learn, TensorFlow та Keras:
- Розробка моделей для класифікації та регресії.
- Створення нейронних мереж для задач розпізнавання образів.

3. Приклади проектів та практичних завдань

#Приклад 1: Аналіз даних з Pandas

```
import pandas as pd

# Зчитування даних з CSV-файлу
data = pd.read_csv('data.csv')

# Очищення даних
data.dropna(inplace=True)

# Аналіз даних
summary = data.describe()

# Візуалізація даних
data.plot(kind='bar', x='Category', y='Value')
```

В цьому прикладі показано, як зчитувати, очищати, аналізувати та візуалізувати дані за допомогою Pandas.

#Приклад 2: Створення веб-додатку з Flask

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
```

```
    return render_template('index.html')

if __name__ == '__main__':
    app.run(debug=True)
```

Цей приклад демонструє основи створення веб-додатку з використанням Flask.

#Приклад 3: Автоматизація роботи з файлами

```
import os

# Створення папки
os.mkdir('new_folder')

# Переміщення файлів
for file in os.listdir('source_folder'):
    os.rename(f'source_folder/{file}', f'new_folder/{file}')
```

В цьому прикладі показано, як автоматизувати створення папок та переміщення файлів.

#Приклад 4: Розробка моделі машинного навчання

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

# Завантаження даних
data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target, test_size=0.2)
```

```
# Розробка моделі
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Оцінка моделі
accuracy = model.score(X_test, y_test)
print(f'Accuracy: {accuracy}')
```

Цей приклад демонструє створення простої моделі машинного навчання для задачі класифікації.

4. Рекомендації щодо подальшого самостійного навчання та розвитку у програмуванні на мові Python

#Вибір ресурсів для навчання

- Книги: "Python Crash Course" від Eric Matthes, "Automate the Boring Stuff with Python" від Al Sweigart.
- Онлайн-курси: Coursera, edX, Udemy.
- Документація: Офіційна документація Python (python.org), документація бібліотек (Pandas, NumPy, Django).

#Поради щодо практики програмування

- Регулярно вирішуйте задачі на платформах, таких як LeetCode, HackerRank, Codewars.
- Участь у хакатонах та конкурсах програмування.
- Практикуйтеся у написанні власних проектів.

#Залучення до спільнот та участь у конкурсах

- Приєднуйтесь до онлайн-спільнот на форумах, таких як Stack Overflow, Reddit (r/learnpython).
- Участь у конференціях та мітапах.
- Беріть участь у конкурсах програмування та хакатонах, таких як Kaggle.

5. Висновок

Використовуючи знання з Python, ви можете реалізувати різноманітні проекти в багатьох сферах. Важливо постійно розвиватися, навчатися новим методам та інструментам, і практикуватися у програмуванні.

Питання для перевірки знань по темі лекції

1. Які основні завдання можна вирішувати за допомогою обробки даних в Python?
   – Правильна відповідь: Аналіз даних, очищення даних, візуалізація даних, підготовка даних для моделювання.

2. Які бібліотеки Python ви знаєте для роботи з обробкою даних?
   – Правильна відповідь: Pandas, NumPy, Matplotlib, Seaborn.

3. Як можна використати Python для автоматизації рутинних завдань? Наведіть приклади.
   – Правильна відповідь: Автоматизація роботи з файлами та папками (створення, перейменування, переміщення), автоматизація збору даних з веб-сайтів (web scraping), автоматизація регулярних звітів.

4. Які фреймворки ви знаєте для веб-розробки на Python? Які їх основні переваги?

– Правильна відповідь: Django (повноцінний фреймворк з усіма необхідними інструментами), Flask (легкий мікрофреймворк для швидкої розробки простих веб-додатків).

5. Як можна використовувати Python для створення моделей машинного навчання?

– Правильна відповідь: Використання бібліотек, таких як Scikit-Learn, TensorFlow, Keras для розробки, навчання та оцінки моделей класифікації, регресії та нейронних мереж.

6. Опишіть процес створення веб-додатку з використанням Flask.

– Правильна відповідь: Встановлення Flask, створення файлу додатку, визначення маршрутів (routes) та функцій обробки запитів, запуск додатку на локальному сервері.

7. Які типи даних можна обробляти з використанням бібліотеки Pandas?

– Правильна відповідь: Табличні дані, дані з CSV, Excel файлів, SQL баз даних, JSON файлів.

8. Які основні етапи розробки моделі машинного навчання?

– Правильна відповідь: Завантаження та підготовка даних, розділення даних на навчальні та тестові набори, вибір та налаштування моделі, навчання моделі, оцінка точності моделі.

9. Які інструменти та методи ви використовуєте для візуалізації даних в Python?

– Правильна відповідь: Бібліотеки Matplotlib, Seaborn для створення різних типів графіків та діаграм, Pandas для вбудованої візуалізації.

10. Опишіть приклад автоматизації роботи з файлами за допомогою Python.

– Правильна відповідь: Наприклад, створення папки, переміщення файлів з однієї папки в іншу, перейменування файлів на основі певних правил або шаблонів, видалення файлів за певними критеріями.

11. Які ресурси ви можете використовувати для подальшого вивчення Python?

– Правильна відповідь: Книги ("Python Crash Course" від Eric Matthes, "Automate the Boring Stuff with Python" від Al Sweigart), онлайн-курси (Coursera, edX, Udemy), офіційна документація Python (python.org), блоги та форуми.

12. Які книги ви б порекомендували для вивчення Python?

– Правильна відповідь: "Python Crash Course" від Eric Matthes, "Automate the Boring Stuff with Python" від Al Sweigart, "Fluent Python" від Luciano Ramalho.

13. Які онлайн-курси є корисними для вивчення Python?

– Правильна відповідь: Курси на Coursera, edX, Udemy, Codecademy.

14. Чому важливо брати участь у конкурсах програмування та хакатонах?

– Правильна відповідь: Це допомагає покращити навички програмування, отримати практичний досвід, працювати в команді, вирішувати реальні задачі та знайти однодумців.

15. Як можна практикувати програмування на Python, використовуючи онлайн-платформи?

– Правильна відповідь: Вирішувати задачі на LeetCode, HackerRank, Codewars, брати участь у змаганнях на Kaggle.

16. Чому важливо використовувати офіційну документацію для вивчення Python та його бібліотек?

– Правильна відповідь: Офіційна документація містить найактуальнішу та достовірну інформацію, приклади використання, опис методів та функцій, рекомендації щодо найкращих практик.

17. Як участь у спільнотах програмістів може сприяти вашому розвитку?

– Правильна відповідь: Взаємодія з іншими програмістами, отримання порад та допомоги, обмін досвідом, участь у проектах з відкритим кодом, доступ до новин та трендів у сфері програмування.

18. Які блоги та форуми ви б порекомендували для вивчення Python?

– Правильна відповідь: Stack Overflow, Reddit (r/learnpython), Real Python, Python Weekly.

19. Як ви можете використовувати власні проекти для покращення своїх навичок програмування?

– Правильна відповідь: Розробка та реалізація власних проектів дозволяє застосовувати теоретичні знання на практиці, вирішувати реальні проблеми, отримувати нові знання та досвід.

20. Які поради ви б дали для подальшого самостійного навчання та розвитку у програмуванні на Python?

– Правильна відповідь: Регулярно практикуватися, брати участь у конкурсах та хакатонах, читати книги та статті, проходити

онлайн-курси, використовувати офіційну документацію, приєднуватися до спільнот програмістів, створювати та підтримувати власні проекти, аналізувати та вчитися на чужому коді.

Література

1. Васильєв О. Програмування мовою Python : навч. посіб. / А. П. Бойко. – Київ : Вид-во Навчальна книга Богдан, 2019. – 504 с.
2. Івановський О.А, Парненко В.С Інформатика. Програмування на PYTHON [Електронний ресурс] : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Конструювання та дизайн машин» спец. 131«Прикладна механіка» / О.А. Івановський, В.С. Парненко, ; КПІ ім. Ігоря Сікорського. –. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2023. – 232 с.
3. Основи інформатики та програмування. [Електронний ресурс] : конспект лекцій : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Комп'ютерні технології в біології та медицині» спец. 122 «Комп'ютерні науки» / КПІ ім. Ігоря Сікорського; уклад.: І. В. Федорін; – Електрон. текст. дані (1 файл). – Київ: КПІ ім. Ігоря Сікорського, 2023. – 179 с.

ЗМІСТ

Навчальне видання

ПРОГРАМУВАННЯ ТА АЛГОРИТМІЗАЦІЯ

Конспект лекцій
для студентів спеціальності 172 «Електронні комунікації та радіотехніка» та 122 «Комп'ютерні науки»
денної та заочної форм навчання

Укладач САВЧЕНКО Микола Володимирович

Відповідальний за випуск проф. Касілов О.В.
Роботу до видання рекомендував проф. Пустовойтов П. Є.
В авторській редакції

План 2025 р, поз. 567

Підп. до друку 2025 р. Гарнітура Times New Roman. Ум. друк. арк. 5,3.
Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2
Електронне видання