НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет прикладної математики
Кафедра системного програмування і спеціалізованих комп'ютерних систем

«На правах рукопису»
УДК _ 004.05_____

До захисту допущено:
Завідувач кафедри
_____Віталій РОМАНКЕВИЧ
«___»_____2025 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Системне програмування та спеціалізовані комп'ютерні системи»

зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Методи оптимізації естетичної та функціональної складової коду програм»

Виконав:
студент ІІ курсу, групи КВ-31мн
Ковтун Святослав Васильович _____

Керівник:
к.т.н, доцент кафедри СП і СКС _____
Клятченко Ярослав Михайлович

Рецензент: _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 року

Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність – 123 «Комп‘ютерна інженерія»

Освітньо-наукова програма «Системне програмування та спеціалізовані комп‘ютерні системи»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

(підпис)

«____» _____ 2023 р.

ЗАВДАННЯ

на магістерську дисертацію студенту

Ковтун Святославу Васильовичу

1. Тема дисертації «Методи оптимізації естетичної та функціональної складової коду програм», науковий керівник дисертації Клятченко Я.М, к. т. н. доцент кафедри СП і СКС, затверджені наказом по університету від 21 березня 2025 р. №1219-с

2. Термін подання студентом дисертації 15 травня 2025 року.

3. Об’єкт дослідження: методи оптимізації швидкодії та естетичного аспекту програм написаних мовою Python

4. Вихідні дані: наявні аналоги комплексних систем оптимізації коду

5. Перелік завдань, які потрібно розробити: запропонувати уніфіковану, ефективну та зручну систему для оптимізації, як буде гарним доповненням до середовища програмування та розширити її новими методами оптимізації коду.

7. Орієнтовний перелік публікацій: тези конференції та стаття

8. Дата видачі завдання

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів дисертації	Примітка
1.	Вивчення літератури за тематикою дисертації	01.02.2024	
2.	Розроблення та узгодження технічного завдання	08.07.2024	
3.	Аналіз існуючих рішень	01.10.2024	
4.	Підготовка матеріалів першого розділу	08.12.2024	
5.	Підготовка матеріалів другого розділу	15.03.2025	
5.	Підготовка матеріалів третього розділу	22.04.2025	
6.	Підготовка матеріалів четвертого розділу	28.04.2025	
7.	Оформлення документації магістерської дисертації. Перевірка на плагіат	01.05.2025	
8.	Попередній розгляд магістерської дисертації на кафедрі	07.05.2025	

Студент Святослав КОВТУН

Науковий керівник дисертації Ярослав КЛЯТЧЕНКО

РЕФЕРАТ

Актуальність теми. У сьогоденні світу створення програмного забезпечення, якість та швидкість виконання коду стають визначальними факторами для підтримки, розширення функціоналу та адаптації програмних продуктів. Однак використання існуючих інструментів, таких як бібліотеки для оптимізації коду, є ускладненим в зв'язку з їх розпорошенням та відсутності єдиного уніфікованого програмного забезпечення. Також наявна проблема недостатньої кількості самого інструментарію. Тому розробка нових та об'єднання з наявними методами для оптимізації естетичної та функціональної складової коду, які враховують різні випадки, є актуальною та важливою задачею, як з наукової, так і з практичної точки зору.

Об'єкт дослідження: процеси та засоби оптимізації програмного коду, зокрема написаного мовою програмування Python.

Предмет дослідження: алгоритми та програмні рішення для автоматизованої трансформації, аналізу та оптимізації коду Python із використанням інструментів форматування, підвищення продуктивності та адаптації до командних стандартів.

Мета роботи: підвищення ефективності засобів для оптимізації (естетичної та продуктивної складових) програмного коду на Python та забезпечення можливості його адаптування до стандартів та потреб розробників програмного забезпечення.

Наукова новизна: запропоновано новий комплексний підхід до оптимізації коду, що поєднує автоматичне форматування, аналіз та покращення продуктивності, генерацію коментарів та адаптацію до нестандартизованих стилістичних вимог команд. Також, вперше реалізовано зворотну трансформацію синтаксису з блокової стилізації до Python-формату та навпаки.

Практична цінність отриманих результатів полягає в тому, що:

Розроблене програмне забезпечення дозволяє значно зменшити витрати часу на підтримку якості коду, покращити читабельність, підтримку та

продуктивність програм Python, а також спростити перехід розробників із інших мов програмування. Система легко інтегрується у робочі процеси команд розробки, завдяки наявності налаштовуваних пресетів.

Апробація роботи:

Клятченко, Я. М., Ковтун С. В. “Методи оптимізації функціональної та естетичної складової коду програм”, Прикладна математика та комп’ютинг ПМК 2024 : збірник тез доповідей Сімнадцятої конференції магістрантів та аспірантів (20-22 листопада 2024 р. Київ, Україна). – Київ, 2024. – С. 558-561.

Клятченко, Я., & Ковтун, С. (2025). МЕТОДИ ОПТИМІЗАЦІЇ ЕСТЕТИЧНОЇ ТА ФУНКЦІОНАЛЬНОЇ СКЛАДОВОЇ ПРОГРАМ. Вісник Національного технічного університету «ХПІ». Серія: Системний аналіз, управління та інформаційні технології, (1 (13) 2025 (прийнято до друку).

Структура роботи. Магістерська робота складається з вступу, чотирьох розділів, висновків, списку літератури та додатків.

У вступі обґрунтовано актуальність теми, сформульовано мету, задачі та наукову новизну дослідження.

У першому розділі розглянуто теоретичні основи оптимізації коду, включаючи поняття якості коду та стандарти кодування та проаналізовано наявний інструментарій.

У другому розділі описується мета та інструментарій, який буде використовуватися для розробки програмного забезпечення.

У третьому розділі спроектовано саму систему, інтерфейс та алгоритми.

У четвертому розділі опис новоствореного програмного продукту та перевірка на відповідність поставленим вимогам.

У висновках узагальнено результати дослідження та наведено рекомендації для подальшого впровадження.

Робота містить 86 сторінок, 44 рисунків, 5 таблиць та 17 джерел літератури.

Ключові слова: оптимізація коду, естетична якість, функціональність, нестандартизовані випадки, PEP 8, автоматизація, інструменти.

ABSTRACT

Relevance of the topic. In today's world, software creation, code quality and execution speed are becoming determining factors for supporting, expanding the functionality and adapting software products. However, the use of existing tools, such as libraries for code optimization, is complicated due to their dispersion and the lack of a single unified software. There is also a problem of insufficient number of tools themselves. Therefore, the development of new and combining with existing methods for optimizing the aesthetic and functional component of the code, which take into account various cases, is a relevant and important task, both from a scientific and practical point of view.

Object of research: processes and tools for optimizing software code, in particular written in the Python programming language.

Subject of research: algorithms and software solutions for automated transformation, analysis and optimization of Python code using formatting tools, increasing productivity and adapting to command standards.

The purpose of the work: to increase the efficiency of tools for optimizing (aesthetic and productive components) of Python program code and to ensure its adaptation to the standards and needs of software developers.

Scientific novelty: a new comprehensive approach to code optimization is proposed, combining automatic formatting, analysis and improvement of productivity, generation of comments and adaptation to non-standard stylistic requirements of teams. Also, for the first time, the reverse transformation of syntax from block styling to Python format and vice versa was implemented.

The practical value of the results obtained is that:

The developed software allows to significantly reduce the time spent on maintaining code quality, improve readability, support and productivity of Python programs, as well as simplify the transition of developers from other programming languages. The system is easily integrated into the workflows of development teams, due to the presence of customizable presets.

Approbation of the work:

Klyatchenko, Ya. M., Kovtun S. V. “Methods for optimizing the functional and aesthetic component of program code”, Applied Mathematics and Computing PMK 2024: collection of abstracts of the Seventeenth Conference of Master's and PhD Students (November 20-22, 2024, Kyiv, Ukraine). – Kyiv, 2024. – pp. 558-561.

Klyatchenko, Ya., & Kovtun, S. (2025). METHODS FOR OPTIMIZING THE AESTHETIC AND FUNCTIONAL COMPONENT OF PROGRAMS. Bulletin of the National Technical University “KhPI”. Series: System Analysis, Management and Information Technologies, (1 (13) 2025 (accepted for publication).**Structure of the work.** The master's thesis consists of an introduction, four sections, conclusions, a list of references and appendices.

The **introduction** substantiates the relevance of the topic, formulates the goal, objectives and scientific novelty of the research.

The **first section** considers the theoretical foundations of code optimization, including the concept of code quality and coding standards, and analyzes the existing tools.

The **second section** describes the goal and tools that will be used for software development.

The **third section** designs the system itself, the interface and algorithms.

The **fourth section** describes the newly created software product and checks for compliance with the requirements.

The **conclusions** summarize the research results and provide recommendations for further implementation.

The work contains 86 pages, 44 figures, 5 tables and 17 references.

Keywords: code optimization, aesthetic quality, functionality, non-standardized cases, PEP 8, automation, tools.

ЗМІСТ

СПИСОК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

AST — абстрактне синтаксичне дерево: структура інформації, що репрезентує код у вигляд розгалужень операцій

CPU — центральний процесор: "мозок" комп'ютера, який оперує з арифметико-логічними операціями

Cython — оптимізуючий компілятор: модифікує Python-код у високошвидкісні C-розширення

JIT — компіляція "на льоту": динамічна генерація машинного коду під час роботи програми

Numba — JIT-компілятор для Python: форсує обчислення через декоратори (@njit / @jit)

NumPy — бібліотека для наукових обчислень: надає удосконалені масиви та математичні методи

ПЗ — програмне забезпечення: набір інструкцій для здійснення задач на комп'ютері

RAM — оперативна пам'ять: стрімке тимчасове сховище даних для процесора

IT — інформаційні технології: сфера зі створення, обробки та передачі електронних даних

ВСТУП

Серцем будь-якої цифрової системи можна вважати програмний код. Сучасний етап розвитку інформаційних технологій характеризується стрімким ускладненням програмного забезпечення, що зумовлює нові вимоги до його ефективності, продуктивності та зручності супроводу. Одним з ключових викликів у цьому контексті виступає оптимізація програмного коду, яка дозволяє прискорити роботу додатків, зменшити споживання ресурсів та покращити читабельність коду для полегшення подальшої розробки та підтримки.

Метою цієї магістерської дисертації є систематизація вже відомих підходів до оптимізації програмного коду, їх аналіз та визначення обмежень, а також розробка нових методів, спрямованих на підвищення читабельності та продуктивності коду програмного забезпечення. Особлива увага приділяється інтеграції цих методів в єдиний програмний інструмент, що дозволить спростити процес оптимізації для розробників різного рівня кваліфікації.

Актуальність дослідження підкреслюється потребою у створенні універсальних рішень, які комбінують переваги різних підходів до оптимізації (наприклад, покращення алгоритмів, використання сучасних бібліотек, паралельні обчислення тощо) та адаптуються до специфіки конкретних завдань. Особливо важливим є забезпечення балансу між продуктивністю та зручністю супроводу коду, що є критично важливим для тривалої експлуатації програмних систем.

Практична цінність дослідження полягає у розробці програмного забезпечення, яке можна буде використовувати для оптимізації реальних проєктів, зокрема у сферах веб-розробки, наукових обчислень та вбудованих систем. Передбачається, що запропоновані методи дозволять скоротити час виконання програм та спростити процес їх подальшого масштабування.

Структура роботи передбачає огляд літератури з проблеми оптимізації коду, опис методології дослідження, аналіз існуючих інструментів, представлення власних розробок. Результати дослідження будуть корисними для

фахівців у галузі програмної інженерії, розробників ПЗ та науковців, які займаються дослідженням алгоритмів та високопродуктивних обчислень.

РОЗДІЛ 1 АНАЛІЗ НАЯВНОГО ІНСТРУМЕНТАРІЮ ТА ТЕОРЕТИЧНИХ ВІДОМОСТЕЙ ПРО ОПТИМІЗАЦІЮ КОДУ МОВОЮ PYTHON

1.1 Визначення поняття «оптимізація коду»

Оптимізація коду – це процес удосконалення програмного коду для підняття його ефективності, зручності використання, продуктивності та відповідності критеріям розробки. У широкому сенсі оптимізацію слід розуміти як вибірка дій, які роблять код не тільки жвавішим для виконання, але й чистішим, більш систематизованим та зрозумілим для інших розробників.

Серцем будь-якої системи є програмний код. Зважаючи, наскільки добре він написаний, залежить не тільки поводження програми під час запуску, але й те, наскільки гладко її можна підтримувати, ампліфікувати та адаптувати до нових вимог. У невеликих проектах це скоріш за все може бути не критично, але у обширних системах навіть незначна неефективність або невідповідність може призвести до великих втрат продуктивності, помилок у роботі та збільшення витрат на підтримку. Саме тому оптимізація коду є важливим компонентом як на етапі розробки, так і в подальшому життєвому циклі програмного забезпечення.

Варто розуміти, що оптимізація не обмежується лише швидкістю. Звісно, найчастіше під цим терміном мають на увазі покращення ефективності: прискорення роботи програм, мінімізацію споживання оперативної пам'яті чи інших ресурсів. Але сучасна оптимізація охоплює набагато ширший спектр задач. Наприклад, вона включає усунення надлишкового коду, введення єдиного стилю кодування, автоматичне форматування, реорганізацію функцій та класів на структурному рівні, поліпшення найменувань змінних, видалення недосяжного коду, а також забезпечення кращої документації. Усі ці дії безпосередньо впливають на легкість супроводу проєкту.

Ще один вагомий момент - збереження балансу в контексті оптимізації. Не кожне вдосконалення є вигідним з огляду на кінцеві результати. Наприклад,

надмірне прагнення до оптимізації, що ускладнює логіку, здатне викликати явище «передчасної оптимізації». Це коли розробник ускладнює код, не спостерігаючи помітного покращення на поточному етапі розробки. Таким чином, оптимізація повинна бути обдуманою, беручи до уваги як потреби, так і реальні обмеження конкретної системи.

Загалом, оптимізація коду – це не одноразова дія, а безперервний процес, що супроводжує розробку від самого початку до її завершення. Вона вимагає розвиненого аналітичного мислення, розуміння внутрішньої структури програми, а також знання особливостей мови програмування та використаних технологій. Ефективне впровадження оптимізації дає змогу створювати не тільки швидкий, а й зручний, зрозумілий та готовий до масштабування код, який буде легко підтримувати та вдосконалювати в майбутньому.

1.2 Основні цілі оптимізації

Оптимізація коду в будь-якому програмному проєкті має єдину мету – покращити його. Втім, розуміння "покращення" залежить від багатьох факторів, таких як контекст, тип застосунку, обмеження середовища виконання та етап розробки. Відповідно, цілі оптимізації розподіляються на кілька ключових напрямків, кожен з яких має практичне значення.

Першим і найчастіше згадуваним задумом є прискорення роботи. Це означає, що потенційно програма може працювати швидше, втрачаючи менше часу на обчислення або обробку даних. У комплексних заплутаних системах або програмах реального часу навіть декілька наносекунд можуть мати критичне значення. Наприклад, веб-сайт, який тривало завантажується, може проґавити користувача ще до того, як відчиниться сторінка. У таких випадках розробник шукає вузькі місця – ділянки коду, яким надали недостатньо уваги і вони сповільнюють роботу, – та оптимізує їх, використовуючи кращі алгоритми або перерозподіляючи обчислення.

Друга не менш важлива ціль – зменшення виснаження ресурсів. Це потенційно може застосовуватися до пам'яті, процесора, диска або мережі. У

оточеннях з обмеженими ресурсами, наприклад - мобільні пристрої або вбудовані системи, навіть дріб’язкове перевищення ліміту може викликати зависання або збої. Удосконалений код потенційно може виконувати ті ж завдання, але користуватися меншим об’ємом системних ресурсів. Це є вагомим також з точки зору енергоефективності, де мова йде про батареї або потужні серверні інфраструктури.

Третім задумом є підвищувати показники читабельності та зрозумілості програмного коду. Хоча це на пряму не впливає на те, як програма працює, добре зрозумілий код сприяє пришвидшенню в виявленні помилки, полегшує внесення змін та посилює командну роботу. У комплексних проектах, над кодом яких трудяться сотні або десятки спеціалістів, об’єднаний стиль оформлення, розбірлива структура функцій та гармонійна послідовність мають вирішальну роль. Оптимізація у цьому випадку інтерпретується як впорядкування, форматування, коментування та стандартизацію коду, з метою перетворення його не лише в коректний, але й зрозумілий для всіх членів команди.

Іншою метою є полегшення супроводу та розвитку проекту в майбутньому. Оптимізований код здебільшого має в собі менше дублювань, він більш компактніший, зрозуміло розділений на модулі та легше розширюється. Коли код вільний від надмірностей, він краще коригується до змін, наприклад, при переміщенні на нову бібліотеку чи платформу. Це дозволяє ухилятися ситуацій, коли технічний борг сповільнює та перешкоджає подальшу роботу над продуктом, зводячи нанівець намагання команди розробників.

Крім того, оптимізація часто направлена на редукцію кількості помилок. Добре сформований та "чистий" код легше тестувати, оскільки наявно менше прихованих взаємозв’язків та заплутаних умов. Систематизовані інструменти оптимізації, наприклад, спроможні видаляти "мертвий" код, що міг би потенційно спричинити до конфліктів або хаотичної поведінки. Таким чином, підвищується міцність всього програмного продукту, забезпечуючи стійку роботу на всіх рівнях.

На підсумок, оптимізація може бути механізмом для забезпечення масштабованості. Це означає, що програма мусить справлятися зі нарощуванням обсягу даних або суми користувачів без серйозного зниження продуктивності. Оптимізований код краще адаптований до посилення навантаження, оскільки вже зважає на використання ефективних структур даних та алгоритмів, що дає спроможність системі зберігати значну продуктивність навіть при зростанні необхідності.

Отже, оптимізація - це не суто про швидкість, а й про якість, надійність, варіативність і довговічність програмного забезпечення. Вона дозволяє створювати системи, які виконують обов’язки ефективно не тільки наразі, але й перебувають актуальними та життєздатними в майбутньому, відповідаючи на запити сучасних технологій.

1.3 Класифікація видів

Оптимізацію коду можна теоретично поділити на категорії, відносно до конкретної мети, яку прагне реалізувати розробник. В цілому, всі форми оптимізації налаштовані на поліпшення якості програмного забезпечення, проте першочергові завдання можуть суттєво відрізнятися: деякі докладають зусиль, щоб досягнути максимального темпу виконання, хтось акцентує увагу на легкості читання коду, а хтось фокусується на зручність для команди розробників. Відтак, для організованого підходу надзвичайно важливо розуміти фундаментальні категорії оптимізації та усвідомлювати, як всяка з них впливає на остаточний результат.

Оптимізація продуктивності

Цей вид оптимізації акцентований на темпі та ефективності роботи програми в реальному середовищі. Вона зачіпає все, що веде до редукції часу виконання, зниження навантаження на CPU або скорочення обсягу пам'яті, який програма споживає під час своєї роботи.

Оптимізація швидкодії переважно включає такі дії:

Заміна неефективних алгоритмів на більш раціональні. Наприклад, перехід від алгоритму з квадратичною складністю до лінійного еквівалента.

Кешування здобутих даних. Цей підхід стає вкрай корисним, коли застосовується рекурсія, або коли функції запускаються численні рази з тими самими заданими параметрами.

Анулювання "мертвих" фрагментів коду – функцій, змінних чи блоків, що не приймають участь у виконанні та гублять обчислювальні потужності.

Розгортання банальних циклічних конструкцій. Це дає змогу інтерпретатору не розтрачувати зайві ресурси на множинне виконання одних і тих же керуючих структур.

Завдяки цій модернізації, програма виразно "легшає", оперативно рефлексує на команди та запити користувача, а також може переробляти значно об’ємніші обсяги даних без редукції ефективності. Це кардинально важливо для мобільних додатків, серверів з величезним навантаженням або систем, призначених для вивчення даних.

Естетичне та стилістичне вдосконалення

Другий принциповий аспект — покращення графічного вигляду, структури та логіки коду, які неопосередковано не впливають на темп виконання, але кардинально покращують якість та комфортність сприйняття коду. Таким чином його помітно простіше читати, супроводжувати, помічати помилки або запроваджувати коригування. Це винятково актуально в командній розробці, коли над єдиним проектом оперує декілька розробників.

Основні складові естетичного вдосконалення включають:

Самостійне форматування коду. Наприклад, балансування відступів, підтримка стандартів PEP 8 у Python, ліквідація зайвих пробілів чи безвмістних рядків.

Гармонізація стилю найменування. Скажімо, реалізація єдиного формату: snake_case або CamelCase — відповідно до потенційних ухвалених в команді стандартів.

Усунення повторюваного коду. Якщо один і той же шматок коду трапляється кілька разів, розумно перекласти його в окрему функцію.

Це кардинально знижує шанс на помилки, які в змозі виникнути через неуважність або безлад з іменами та структурою. Крім того, такий код стає доступнішим для передачі іншому програмісту або для підтримки протягом тривкого часу після написання.

Отже, пара аспектів - ефективність та краса - споріднені. Один гарантує технічну продуктивність програми, інший – її структурну зрозумілість та зручність для програміста. Нехтування будь-яким із цих факторів веде до проблем: або програма працює повільно, або ж вона настільки складна, що кожна зміна стає потенційно ризикованою.

Сучасні механізми надають можливість комбінувати ці підходи - шляхом впровадження в єдину програму. Саме таке сполучення було втілено в розроблене програмне забезпечення для всебічної оптимізації Python-коду.

1.4 Естетична складова коду

Коли ми обмірковуємо над естетикою коду, мова не про екстер'єрну привабливість чи витонченість. Натомість, йдеться про глибинну злагодженість, логіку та синхронність, що роблять програму легкою для прочитання, сприйняття та підтримки. Зокрема кажучи, у Python, який від самого початку був спроектований з наголосом на чистий, зрозумілий синтаксис, естетика виконує визначально роль. Добре написаний Python-код читається ледь не як звичайний текст, обходячи зайвої плутанини чи неоднозначного трактування.

Естетичний напрям містить декілька аспектів, і початковим з них є стиль оформлення. Це включає коректні відступи, балансування рядків, зрозумілу будову умов, циклів, функцій та класів. У обраній мною мові відступи – це не просто дилема візуального сприйняття, а сегмент синтаксису. Проте те, як саме творець організовує ці блоки, часто репрезентує його підхід до читабельності. Як варіант, код з еквівалентними відступами, з логічно організованими рядками, без

турбулентних переносів – завжди легше розтлумачити, навіть якщо ти не є автором.

Другим основоположним аспектом є регулювання назв. У Python існують традиційні правила: змінні доцільно називати у за технологією «змійки», класи — у спосіб «верблюда», а константи — у за методикою великих літер. Якщо безупинно дотримуватися цих звичок, код буде сприйматися знайомим кожному Python-розробнику, навіть якщо він розглядає його вперше. А от несистематичне упровадження стилів, схожих на lowUP, user_info, LowUP, породжує емоцію хаосу, погіршує розуміння завдань об'єктів та несприятливо впливає на потенційну підтримку.

Ще один визначальний аспект — існування та якість коментарів. Мова не просто про те, щоб "щось викласти над функцією", а про те, щоб розкрити суть складного, виправдати прийняті рішення, вказати на рамки чи винятки. Унікальне значення мають docstrings – професійні коментарі до функцій, класів або модулів, які моментально описують їхню суть. Відмінний коментар не дублює код, а додає сутності: він пояснює "чому", а не "що".

Не менш ключовим є логічна організація коду. Естетика – це також про те, щоб не комбінувати логіку в один великий клубок, а розмежовувати її між функціями, методами та модулями. Один сектор коду — одна компетенція. Це спрощує орієнтування кодом, дає змогу впроваджувати зміни в окремі шматки без ризику пошкодити інші. Як варіант, функція з багатьма рядками коду і розмитою метою виглядає відштовхуюче не через свій розмір, а тому що вона "перевантажена" та нечітка.

Естетична оптимізація також містить ліквідацію дублювання, зайвого коду та хаотичних виразів. Якщо та сама логіка зустрічається дещо більше разів, краще витягнути її в окрему функцію. Якщо вираз можна вікоригувати в простішу форму, це варто зробити (рис. 1.1).

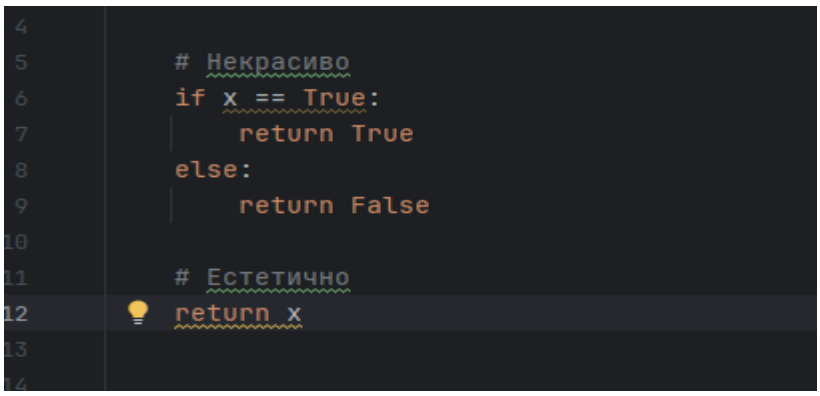


Рисунок 1.1 – Фрагмент коду мовою Python в якому зображено усування заплутаних виразів

У командній роботі естетика на додачу формується єдністю стилю між усіма без вийнятку учасниками. Коли один творець форматує код іншим способом, ніж решта, це створює безладдя у репозиторії, обтяжує злиття гілок, генерує конфлікти, навіть якщо послідовність коду не змінюється. Тому чимало команд використовують автоматичні коректори, які автоматично стабілізують стиль коду відповідно до PEP 8 або внутрішніх стандартів.

І, врешті-решт, естетика в Python — це маніфестація поваги. До себе майбутнього, хто прибуде назад до цього коду через рік. До товаришів по роботі, яким випаде на долю його читати. До самого процесу розробки, де пріоритетно не лише "щоб працювало", а й щоб було реалізовано грамотно та естетично.

Відтак, естетична складова коду — це не вторинна дрібниця, а один з фундаментальних елементів якісної розробки. Вона напряму не впливає на продуктивність програми, але відчутно прискорює процес розуміння, інтеграція змін та підтримки коду. І саме у результаті цього естетика в Python наповнюється глибоким змістом — як з технічної, так і з практичної точки зору.

1.5 Збільшення продуктивності

Під продуктивністю коду мається на увазі його здатність блискавично виконуватися, раціонально оперувати ресурсами комп'ютера (CPU, RAM), а також стабільно виконувати завдання навіть при зростанні емності даних чи

навантаження. Хоча Python не належить до миттєвих мов програмування з погляду фундаментальної оптимізації, він має громіздкий потенціал для покращення продуктивності у результаті грамотної стратегії розробки. У цьому тлумаченні продуктивність – це не лише компетенція компілятора чи інтерпретатора, а і досягнення правильного осмислення програміста.

Основні напрями для нарощення продуктивності:

Виправлення неефективних конструкцій

Повсякчас повільна робота коду є відлунням не мови програмування, а погано сконструйованої логіки. Як варіант, багато хто оперує вкладеними циклами, коли місію можна вирішити за допомогою вмонтованих функцій або структур даних. Це елементарний приклад, хоча він показує, як зміна принципу може урізати час виконання в десятки разів (рис. 1.2).

```
11  for i in range(len(users)):
12      for j in range(len(users)):
13          if users[i] == users[j] and i != j:
14              print("Збіг")
15
16
17  if len(users) != len(set(users)):
18      print("Збіг")
```

Рисунок 1.2 – Фрагменти коду мовою Python в якому зображено приклад заміни підходу

Збереження результатів опрацювання даних

Python представляє корисні інструменти для архівування проміжних результатів операцій, що дозволяє запобігти дубльованого обчислення. Така концепція особливо дорогоцінна у випадках рекурсивних функцій, або у той час як одна й та сама операція викликається чимало разів з тими ж аргументами. Цей

концепція значно прискорює калькуляції, особливо у випадках з об’ємною кількістю викликів (рис. 1.3).

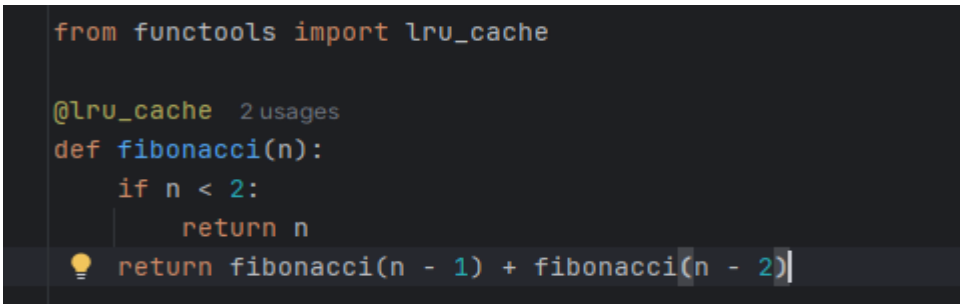


Рисунок 1.3 – Фрагмент коду мовою Python в якому зображено кешування рекурсії

Усунення мертвого та непотрібного коду

Нагромадження зайвих імпортів, неактивних функцій чи змінних, які не формують кінцевий результат, є додатковим навантаження не лише для інтерпретатору, а й на самого творця, котрий вимушений "блукати" у надмірності інформації. Ліквідація такого коду не лише конструктивно впливає на продуктивність, а й посилює загальний рівень якості процесу розробки.

Оптимізація використання структур даних

Зазвичай, заміщення одного типу структури даних іншим може значно збільшити швидкість роботи. Якщо пояснювати мовою прикладів, то доступ до елементів списку потребує більше часу, ніж до словника чи множини, особливо якщо мова йде про пошук даних, але така проста заміна (рис. 1.4) здатна зменшити час перевірки з лінійного до сталого.

Розгортання звичайних циклів

Цикли мають певні витрати на виконання умов, лічильників та управління перемиканнями.

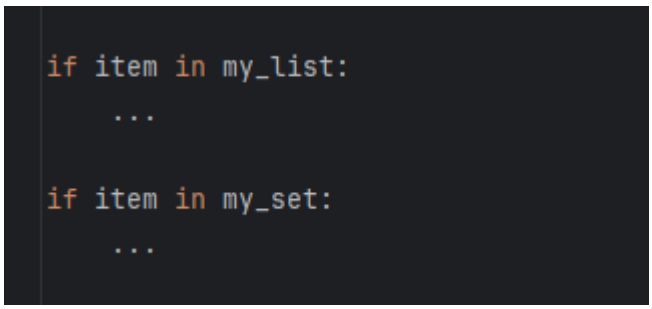


Рисунок 1.4 – Фрагмент коду в якому приклад заміни на більш ефективні конструкції

Якщо кількість повторень визначена і незначна, інколи краще ясно розгорнути цикл (рис. 1.5). Звісно, це слід застосовувати обережно, лише якщо справді є сенс.

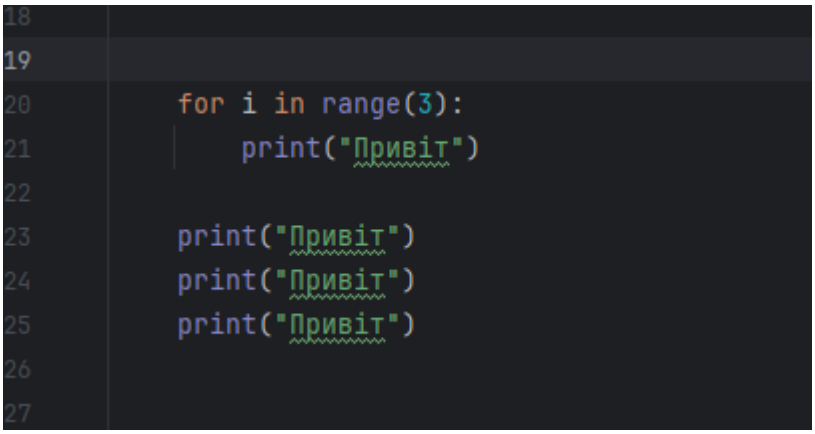
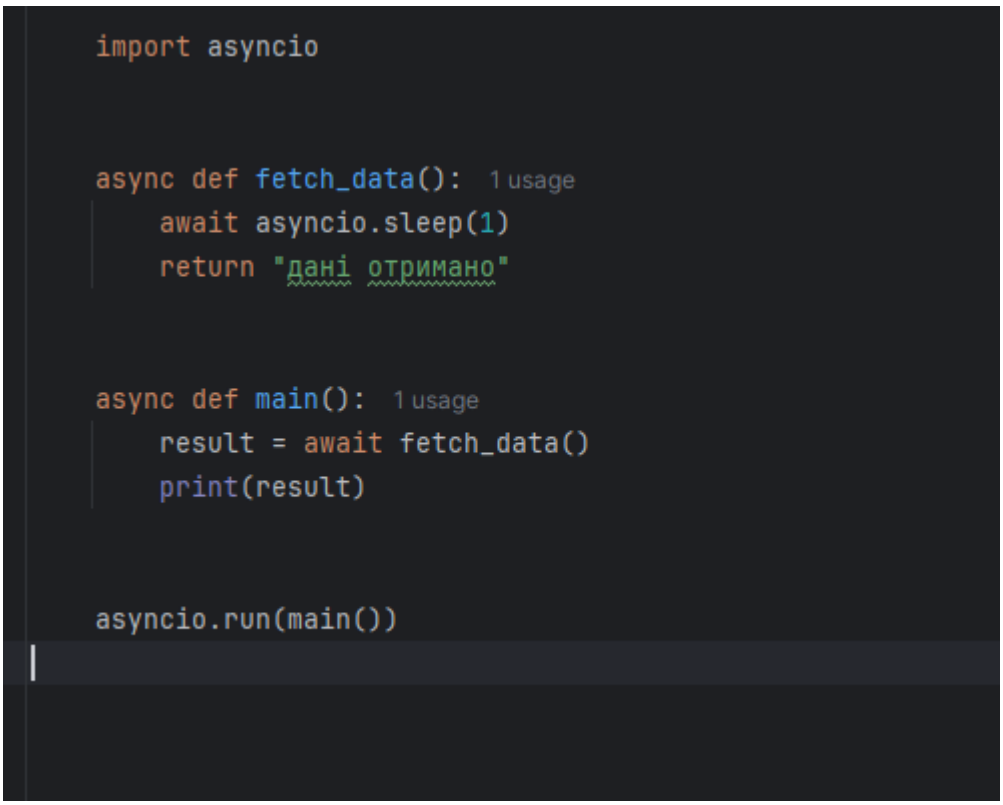


Рисунок 1.5 – Фрагмент коду мовою Python в якому показано розгортання циклів

Асинхронність та багатопоточність

Задля задач, пов’язаних з введенням / виведенням (наприклад, робота з мережею, файлами, API), продуктивність можливо значно поліпшити, застосовуючи asyncio (рис. 1.6) або багатопоточну обробку. Це дає змогу не чекати завершення кожної операції, а виконувати інші завдання паралельно.



```python
import asyncio


async def fetch_data():  1 usage
    await asyncio.sleep(1)
    return "дані отримано"


async def main():  1 usage
    result = await fetch_data()
    print(result)


asyncio.run(main())
```

Рисунок 1.6 – Фрагмент коду мовою Python в якому показано застосування бібліотеки asyncio

Використання вбудованих та C-оптимізованих бібліотек

Можна побачити, шо деякі модулі, часто втілені на рівні C, тому і швидкість обробки в них значно швидше за подібні реалізації на чистому Python. Якщо потрібні обчислення з великими обсягами даних, застосування таких бібліотек — це безпосереднє заощадження часу.

Загальна ідея

Підвищення продуктивності коду Python — це не магія і не завжди пов’язано з низькорівневою оптимізацією. Це насамперед про увагу до дрібниць, розуміння того, як саме відбувається код, і вміння обрати потрібний інструмент або підхід. Навіть невеликі зміни здатні мати помітний вплив на швидкодію, особливо у великих проєктах.

Важливо пам’ятати: завчасна оптимізація — ворог. Не слід оптимізувати усе підряд "про всяк випадок". Спочатку варто мати стабільний і правильний код,

а вже потім — вимірювати його швидкодію та оптимізувати лише ті частини, які дійсно впливають на підсумок.

1.6 Стандарти та інструменти форматування

В процесі написання програмного коду надзвичайно важливо не тільки те, що власне робить програма, а й те, як вона виглядає. В цьому контексті форматування коду — це низка норм, що визначають зовнішній вигляд програми: відступи, пробіли, розміщення дужок, назви змінних, порядок елементів у класах тощо. Форматування безпосередньо не впливає на швидкість виконання програми, але воно помітно впливає на сприйняття, супровід та розвиток проєкту. Особливо це важливо у великих командах, де десятки людей працюють з одним кодом.

Чому важливі стандарти?

Форматування — це свого роду мова між розробниками. Коли всі дотримуються одного стилю, код сприймається легко та природно. Не потрібно витрачати час на здогадки, де починається функція, а де — новий блок логіки. Кожен рядок виглядає так, як очікується. В результаті це зменшує кількість помилок, прискорює розуміння чужого коду та полегшує рев’ю.

У Python одним з найвідоміших стандартів є PEP 8 — набір рекомендацій, що визначає, яким повинен бути чистий і зручний для читання код. Наприклад, PEP 8 регламентує такі речі, як чотири пробіли для відступу (а не табуляція), дві порожні рядки між класами, однаковий стиль назв тощо. Це не обов’язкові правила, але вони є загальноприйнятими, тому більшість інструментів і проєктів орієнтуються саме на них.

Інструменти для автоматичного форматування

Щоб не перевіряти кожен рядок коду вручну, існують спеціальні утиліти, котрі автоматично форматують код відповідно до певних стандартів. Вони аналізують структуру програми та самостійно вирівнюють пробіли, змінюють

порядок складових частин, усувають зайві рядки тощо. Ці інструменти особливо корисні для підтримки єдиного стилю коду у командних проєктах.

Ось декілька прикладів інструментів, що використовуються в Python-розробці:

Black — це унікальний інструмент автоматичного форматування Python-коду, який значно відрізняється від аналогів своїм підходом. Його ключова особливість — мінімум гнучкості в налаштуваннях, що одночасно є його найбільшою перевагою та певним обмеженням [2].

Головна його філософія полягає в тому, що форматер бере на себе всі рішення щодо стилю коду, звільняючи розробників від потреби витрачати час на суперечки щодо дрібниць форматування. Цей підхід має як позитивні, так і негативні сторони, які варто врахувати перед інтеграцією в ваш проєкт.

Переваги Black:

1) Повна однорідність коду

Найважливіша перевага — гарантована єдність стилю в усьому проєкті, незалежно від кількості розробників. Якщо всі використовують Black, код кожного члена команди виглядатиме однаково, що суттєво полегшує його розуміння та читання.

2) Немає витрат часу на налаштування

Інструмент не вимагає жодних налаштувань — він функціонує "з коробки" з оптимальними (на думку розробників) параметрами. Це економить час, який зазвичай витрачається на обговорення та конфігурування стилю кодування.

3) Висока швидкість роботи

Black оптимізований для швидкої обробки навіть великих кодових баз. Форматування відбувається майже миттєво, не сповільнюючи процес розробки.

4) Передбачуваність результату

Ви точно знаєте, як виглядатиме відформатований код, адже інструмент завжди дотримується одних і тих самих правил, без винятків.

5) Актуальні стандарти

Форматер враховує сучасні рекомендації PEP 8 та інші передові практики Python-розробки, регулярно оновлюючись для підтримки нових можливостей мови.

Недоліки Black:

1) Повна відсутність гнучкості

Найбільше обмеження — практично повна відсутність можливостей для налаштування. Наприклад, ви не можете змінити максимальну довжину рядка (вона фіксована на рівні 88 символів) або спосіб вирівнювання аргументів функції.

2) Ігнорування контексту

Інструмент іноді робить код менш читабельним, дотримуючись своїх правил. Особливо це помітно у випадках зі складними структурами даних або матрицями, де ручне форматування було б доречнішим.

3) Складність інтеграції в існуючі проекти

Якщо проект довгий час розвивався з іншим стилем кодування, масове застосування Black може призвести до величезного diff у git, що ускладнить аналіз історії змін.

4) Обмеженість для специфічних випадків

У деяких ситуаціях (наприклад, при роботі з JSON-подібними структурами чи математичними матрицями) строгі правила Black можуть зробити код менш зрозумілим, ніж ручне форматування.

5) Психологічний бар'єр

Деякі розробники можуть сприймати відсутність контролю над форматуванням як обмеження їхньої свободи, що може викликати спротив у команді.

Практичні аспекти використання:

Black особливо добре підходить для:

1) Нових проектів, де ще не сформований стиль кодування
2) Великих команд з багатьма розробниками

3) Проектів з високими вимогами до узгодженості коду

4) Випадків, коли потрібно швидко підняти читабельність існуючого коду

Менш підходить для:

1) Проектів зі специфічними вимогами до форматування

2) Команд, де частина розробників вже має усталені звички

3) Випадків, коли важливе ручне форматування окремих фрагментів

Висновки:

Black — це радикальний, але надзвичайно ефективний інструмент, який може значно підвищити продуктивність команди, усунувши всі суперечки про стиль кодування. Його варто розглядати не просто як форматер, а як інструмент стандартизації, який змінює підхід до написання коду в цілому.

Головне питання перед впровадженням — чи готова команда повністю довіритися автоматизації у питаннях стилю. Якщо так — Black може стати одним з найкорисніших інструментів у будь-якому робочому процесі, але якщо важливо зберігати контроль над деталями форматування — варто розглянути альтернативи з більш гнучкими налаштуваннями.

Autopep8 – це засіб для автоматичного приведення Python-коду до відповідності стандарту PEP 8. На відміну від радикального підходу Black, autopep8 працює більш обережно, надаючи розробникам певну свободу у виборі стилю [1].

Головна перевага autopep8 – його адаптивність. Він не нав'язує жорстких правил, а допомагає привести код до відповідності з рекомендаціями, залишаючи простір для маневру. Інструмент особливо корисний для проектів, де важливо зберегти існуючу структуру коду, але потрібно виправити очевидні помилки форматування.

Ключові особливості autopep8:

Поетапна корекція

Autopep8 дозволяє застосовувати зміни поступово, вибираючи конкретні категорії виправлень. Наприклад, можна виправити лише відступи чи пробіли навколо операторів, не чіпаючи інші елементи.

Конфігурована гнучкість

Користувач може тонко налаштувати рівень агресивності форматування – від мінімальних змін до повного переформатування за стандартами PEP 8.

Збереження авторського стилю

Інструмент намагається мінімізувати втручання в існуючий код, зберігаючи оригінальну структуру там, де це не суперечить основним правилам.

Підтримка часткового форматування

Можна обробляти окремі файли або навіть фрагменти коду, що корисно при поступовій модернізації великих проектів.

Нюанси роботи:

Autopep8 чудово справляється зі стандартними випадками, але може потребувати додаткового налаштування для складних ситуацій. Він не завжди ідеально обробляє довгі ланцюжки викликів або складні вирази, де іноді потрібне ручне втручання.

Цей інструмент особливо корисний для:

1) Початкового наведення порядку в існуючих проектах
2) Команд, які працюють з різними стилями кодування
3) Випадків, коли потрібно підготувати код до code review
4) Навчання правильному форматуванню початківців

Autopep8 – це не диктатор стилю, а розумний помічник, який допомагає підтримувати порядок у коді, не обмежуючи свободу розробників. Він ідеально підходить для проектів, де важливий баланс між стандартизацією та гнучкістю.

YAPF (Yet Another Python Formatter) – це унікальний інструмент для автоматичного форматування, який поєднує підходи Black та autopep8, пропонуючи власний розумний спосіб обробки коду. На відміну від інших форматерів, YAPF не просто сліпо слідує правилам, а аналізує код за допомогою

алгоритмічних методів, наближених до штучного інтелекту, щоб знайти оптимальний стиль форматування [10].

Основні принципи роботи YAPF:

1) Контекстно-залежне форматування

YAPF аналізує структуру коду та його семантику, приймаючи рішення про форматування на основі логічних зв'язків між елементами, а не просто жорстких правил.

2) Глибинне розуміння коду

Форматер здатний розпізнавати шаблони використання та адаптувати стиль під конкретні конструкції, зберігаючи логічну цілісність коду.

3) Гнучкість конфігурації

На відміну від радикального Black, YAPF пропонує численні налаштування, дозволяючи адаптувати стиль під конкретні потреби проекту.

Переваги YAPF:

1) Інтелектуальне форматування

Алгоритми YAPF здатні приймати рішення, які часто виявляються кращими за просте слідування PEP 8, особливо у складних випадках.

2) Баланс між стандартизацією та читабельністю

Форматер знаходить компроміс між суворими правилами та практичною зручністю читання коду.

3) Підтримка різних стилів

Можливість вибору між різними пресетами (Google, Facebook тощо) або створення власного стилю.

4) Збереження логічної структури

YAPF намагається мінімізувати зміни, які можуть ускладнити розуміння коду, на відміну від більш агресивних форматерів.

Недоліки та особливості:

1) Складність передбачення результату

Через використання складних алгоритмів результат форматування не завжди можна точно передбачити.

2) Вища складність налаштування

Багато параметрів можуть ускладнити початкову конфігурацію.

3) Необхідність адаптації

Командам може знадобитися час, щоб звикнути до прийнятих YAPF рішень.

Ідеальні сценарії використання:

1) Для проектів, де важлива як стандартизація, так і збереження читабельності

2) У командах з досвідченими розробниками, які цінують гнучкість

3) Для складних кодових баз з нестандартними конструкціями

4) Коли потрібен баланс між автоматизацією та контролем над стилем

YAPF – це інструмент для тих, хто шукає розумний компроміс між жорсткою стандартизацією Black та обмеженими можливостями autopep8. Він особливо корисний у великих проектах, де важливо зберігати логічну цілісність коду при автоматичному форматуванні.

isort – це спеціалізований інструмент для автоматичного сортування та форматування імпортів у Python-проектах. На відміну від універсальних форматерів коду, isort зосереджений виключно на роботі з імпортами, пропонуючи глибоку та деталізовану обробку цього критично важливого аспекту коду [5].

Філософія isort полягає у тому, що правильно організовані імпорти – це не просто естетика, а важливий інструмент підтримки читабельності та підтримки коду. Інструмент надає розробникам чітку структуру, за якою легко орієнтуватись у залежностях проекту.

Ключові можливості:

1) Інтелектуальна класифікація

Автоматично розподіляє імпорти за категоріями:

1. Стандартні бібліотеки Python
2. Сторонні пакети
3. Локальні модулі проекту

Для кожної групи застосовує окремі правила сортування

2) Гнучкі стратегії сортування

Підтримує різні підходи до впорядкування:

1. Алфавітний порядок
2. Групування за типами (константи, класи, функції)
3. Відповідність стилю конкретних компаній (Google, Facebook)

3) Адаптивне форматування

1. Оптимальне розміщення довгих списків імпортів
2. Автоматичне комбінування імпортів з одного модуля
3. Підтримка коментарів та інструкцій noqa

Переваги використання:

1) Ефективна робота з великими проектами

Особливо корисний для кодових баз з сотнями модулів і складними залежностями

2) Інтеграція з іншими інструментами

Чудово поєднується з Black, flake8 та іншими лінтерами

3) Конфігурована гнучкість

Можливість тонкого налаштування під конкретні вимоги проекту

4) Підтримка альтернативних стилів

5) Може адаптуватись до нестандартних вимог кодстайлу

Типові сценарії застосування:

1) Приведення до ладу хаотичних імпортів у legacy-проектах

2) Підтримка консистентності у великих командах

3) Автоматизація підготовки коду до ревью

4) Інтеграція в pre-commit хуки

5) Використання як інструменту навчання для новачків

Висновок:

isort – це не просто форматер, а потужний інструмент для підтримки порядку в імпортах, який особливо незамінний у професійній розробці. Він

економить години рутинної роботи, запобігає конфліктам і підвищує загальну якість кодової бази. Для Python-розробників, які серйозно ставляться до якості свого коду, isort часто стає таким же обов'язковим інструментом, як і сам інтерпретатор Python.

Практичне значення форматування

Форматування — це не про естетику заради естетики. Воно реально впливає на роботу команди:

1) Зменшує час на читання та рев’ю.
2) Дозволяє швидко знаходити проблемні місця.
3) Полегшує злиття коду з різних гілок.
4) Робить проєкт придатним для відкритої співпраці.
5) Дозволяє автоматизувати перевірку якості коду (lint-аналізатори, CI).

Форматування — це також і дисципліна. Навіть у невеликих проєктах привчання до єдиного стилю формує відповідальне ставлення до коду, що потім переноситься на більш складні задачі. Саме тому форматування є обов’язковим етапом в автоматизованих пайплайнах — перед злиттям коду в основну гілку система перевіряє, чи все відповідає стандарту.

Форматування — це своєрідна «гігієна коду». Воно не замінює логіку і не виправляє помилки, але створює середовище, в якому програмування стає зрозумілішим, швидшим і комфортнішим. І в цьому сенсі воно є невіддільною частиною професійної розробки.

1.7 Інструменти для покращення продуктивності

Покращення продуктивності можна оптимізувати, адже це не завжди зводиться до аналізу та рахування кожного циклу своїми руками. Якщо подивитись на сучасність, то можна побачити, що велика кількість розробників це робить за допомогою деяких бібліотек доступних для всіх користувачів. Головна задача цих чудесних додатків знайти вузькі місця та або автоматично виправити, або показати наявні способи вирішення.

Почнемо з Pylint – це один із самих потужних аналізаторів коду. Його головна задача у ґрунтовній перевірці відповідно до запрограмованих стандартів якості. Якщо порівнювати із простими «лінтерами», описана вище бібліотека не тільки перевіряє на синтаксичні помилки, а й проблеми в архітектурі, можливі шматки, які некоректно виконують роботу та невідповідності стилістиці [6].

З головних особливостей можна окреслити:

1) Аналіз багатьох рівнів
    1. Помилки в синтаксисі
    2. Дублювання в коді
    3. Порушення в стилістиці
    4. Архітектурні проблеми
    5. Помилки логіки коду
2) Гарна система оцінювання
    1. Оцінює за шкалою від 0 до 10
    2. В кінці створюються детальні звіти зі зазначеними шляхами вирішення
    3. Використовує не тільки чисельність, а й відповідне значення для оцінювання
3) Покращені конфігураційні можливості
    1. Потенційна можливість нехтування окремими проблемами
    2. Редагування правил перевірки
    3. Можливість додавання правил користувача

Які в цього інструмента переваги відповідно до конкурентів:

1) Можливість додати до CI/CD

Потенційно важлива ознака для збірки

2) Багатогранна стратегія до забезпечення найкращої якості коду якості коду

Може визначати проблеми які не знаходять навіть після детального ручного аналізу коду, так званого «code review»

3) Ідентифікує можливі баги в коді

Знаходить проблеми до того, як вони стали катастрофічними проблемами, адже як ми всі знаємо чим раніше помилка знайдена тим менше вона буде коштувати на виході.

4) Гарний інструмент для новачків

Формує гарні звички у новачків для аналізу коду

Також коротко про недоліки:

1) Основним є превеликий рівень суворості

Може вимагати великих зусиль для перетворення коду до того який відповідає всім поставленим вимогам

2) Упередженість деяких норм

Частина правил можуть бути дискусійними або взагалі спірними для окремих команд розробки

3) Значний час обробки

Також вважається одним із основних недоліків, адже обробка значних за об’ємом проектів займає пропорційно велику частину часу, але це тільки показує наскільки це потужний інструмент в сучасних реаліях.

Відповідно до моїх думок можна і визначити сценарії використання, а саме:

1) Машинальний перегляд пул-ревестів
2) Переробка «легасі» коду
3) Допомога новачкам
4) Релізна підготовка програмного забезпечення
5) Якісний контроль відповідно до стилів команди

Якщо ж підсумовувати вище зазначене, то можна зрозуміти, що Pylint – є невід’ємною частиною для професіоналів, які використовують Python, однією з головних цілей якого є належного рівня оптимізації коду. Якщо казати про тривалу перспективу, то це чудове програмне забезпечення допомагає підтримувати надійність та забезпечує масштабованість різного роду програмного коду, але не треба забувати і про його високий рівень вибагливості.

Також як зазвичай скажу, що дуже гарно та корисно було б його використовувати з іншими інструментами покращення якості коду.

Flake8 – це зручний та мінімалістичний інструмент, який є одним із найпопулярніших на сучасному ринку та має в собі функціонал багатьох аналізаторів, Він дає нам продуктивний та цілеспрямований розбір коду вимагаючи при цьому невеличкі налаштування, якщо порівнювати з аналогічними рішеннями [4].

Особливості цього продукту:

1) Комбінований погляд на можливості програми, а він складається з таких частин:

    1. PyFlake – відповідає за аналіз помилок в плані логіки програмних рішень.

    2. McCabe – якщо коротко, то аналізує наскільки код є складним.

    3. Pycodestyle – обробка на відповідність всім нам відомому стилю PEP8.

2) Порівняно з бібліотекою яку ми розглядали до цього є значно продуктивним рішенням

Його рівень оптимізації надає можливість швидкої перевірки навіть значних за своєю природою проєктів

3) Структура модульності

Є можливість додання аналогів плагінів для розширення вже наявного функціоналу бібліотеки

Якщо казати про переваги цього програмного забезпечення, то до них можна віднести такі характеристики:

1) Інтеграційні

Має особливість просто інтегруватися з:

    1. «Прекоміт хуками»

    2. CI/CD функціоналом

    3. Також можлива інтеграція з редакторами коду

2) Маштабованість

Як і в попередній бібліотеці наявні інструменти для відключення деяких норм або заборон.

Також є можливість створення своїх правил обробки.

3) Система плагінів

Є широкий асортимент додаткового функціоналу у вигляді бібліотеки плагінів.

4) Легкий для використання

Для початку роботи з бібліотекою є необхідність тільки в мінімальних налаштуваннях.

Простий для розуміння звіт про помилки.

Відповідно до вище зазначеного можна створити сценарії використання цього продукту:

1) Створення у новачків жаги до «best practices» (найкращих практик) мови

2) Автоматизованість в «пул-ревестному» тестуванні

3) Одна стилізація в команді на всіх

4) Можлива постійна обробка коду при розробці

5) Продуктивне оцінювання відповідності по якості коду написаного до сторонніх бібліотек

Якщо, порівнювати з іншими аналогами, то Flake8:

1) Є менш суворішим

2) Більш швидким

3) Більш зрозумілим

4) Більш масштабованим

В кінці підсумуємо, що Flake8 можна вважати гарним варіантом для тих, котрі хочуть рівновагу між швидкодією самого аналізатора та ефективністю оптимізації коду на виході роботи бібліотеки. Він є незамінним інструментом розробників, щоб отримувати аналіз без зайвого навантаження на комп’ютер. А для великої кількості команд я наявним мінімальним стандартом оцінювання коду.

Для нешвидких кодів гарними засобами вимірювання продуктивності є профайлери. Проаналізувавши інтернет можна виділити дві основні бібліотеки такого типу:

1) Yappi. (Yet Another Python Profiler)
2) cProfile

Почнемо з cProfile, якого можна назвати базовим для мови програмування Python, але він надає достатню інформацію, а саме час роботи програми та кількість викликів функцій, які знаходяться в заданому коді. Цей профайлер відрізняється від зовнішніх бібліотек, тим що він вже влаштований в інструментарій мови програмування і готовий до використання одразу після встановлення офіційного програмного забезпечення Python x.xx, де x – це номер версії [3].

Які у цього профайлеру є можливості:

1) Багата на деталі статистика (рис. 1.7)
    1. Час виконання функції
    2. Кожний час виклику функцій
    3. Їх ієрархія
    4. Також кількість

```
      4 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 script.py:26(test_function)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Рисунок 1.7 – Результат роботи профайлеру

2) Різні особливості інтегрування та використання
    1. Через командний рядок
    2. Додавання в код
    3. Переробка самих функцій

3) Незначний вплив на швидкодію

1. Незначні додаткові витрати

2. Потенційно гарна інтеграція з розробкою

Тож у нас є можливість окреслити та визначити головні переваги цього профайлеру:

1) Не потрібно завантажувати за допомогою pip

Додана до списку базових бібліотек Python

2) Уніфікація

Припускається використання до будь якого коду

3) Зручність у використанні

Маленька кількості коду для отримання великої кількості результатів

4) Інтегрованість

Можна використовувати pstats або візуалізовувати для кращого сприйняття

По традиції опишемо сценарії для використання:

1) Порівняння продуктивності алгоритмів

2) Для знайдення повільних за швидкодією місць

3) Допомога новачкам в оптимізації їх перших програм

4) Покращення важливих фрагментів коду

Але також важливо уточнити деякі обмеження:

1) Не демонструє завантаження пам’яті

2) Недостатній влаштований інструментарій візуалізації

3) Не коректні результати для маленьких структур

Також додаю приклад використання до вказаних мною результат роботи програми:

Рисунок 1.8 – Приклад використання cProfile

В кінці хочеться підсумувати, що cProfile є одним із найважливіших інструментів сучасного розробника, його головною метою є знаходження повільних ділянок коду без ускладнених налаштувань. Незважаючи на його маленьку еластичність порівняно з сучасним профайлерами, він підкупає свого користувача наявною простотою та доступністю. Також я повинен підкреслити, що для кращого сприйняття важливо використовувати додаткові інструменти для візуалізації результатів обробки cProfile.

Уявіть ситуацію: ваш код працює, але повільно. Ви вже перепробували звичайні профайлери – вони показують цифри, але не розкривають справжньої причини. Ось тут і починається магія Yappi. Він не просто вимірює час виконання – він розповідає історію вашого коду, розкриваючи приховані сюжетні лінії, які ви навіть не підозрювали.

Асинхронний код? Для Yappi це не проблема.

Коли інші профайлери блукають у лабіринтах asyncio начебто в темній кімнаті, Yappi рухається з упевненістю слідчого з ліхтарем. Він точно знає, які корутини витрачають час на очікування, де ваші await-и перетворюються на

вузькі місця, і навіть покаже, чи потоки дійсно працюють, а не імітують активність [11].

Багатопоточна архітектура? Yappi розбереться.

Ваш код стрибає між потоками, як акробат у цирку? Звичайні інструменти часто втрачають нитку аналізу, але Yappi відстежує кожен перехід, показуючи справжню ціну таких стрибків. Він розкриє, скільки часу кожен потік витрачає на корисну роботу, а скільки – на марне очікування.

Найкраще? Практично нульовий вплив на продуктивність.

Деякі профайлери сповільнюють код настільки, що результати стають марними. Yappi працює настільки обережно, що його можна сміливо використовувати наживо, навіть у продакшен-середовищі. Це як мати детектор брехні, який ніхто не помічає.

Ось як це працює на практиці:

Ви пишете складний мікросервіс, який іноді "тормозить". Звичайні інструменти показують загальні цифри, але Yappi розповість цілу історію: "Ось цей конкретний запит створює чергу, тому що функція X блокується на операції Y, яка, до речі, викликає зайві 30 мс очікування через те, як ви організували потоки".

Чому Yappi – це інший рівень?

Традиційні профайлери – це як термометр: показують температуру, але не говорять, що робити. Yappi – це ціла діагностична система. Він не просто каже "тут повільно", а пояснює: "Це повільно, тому що ви заблокували потік на операції вводу-виводу, і ось як це виправити".

Для кого це?

Для розробників, які не шукають простих шляхів. Для тих, хто хоче не просто знайти проблему, а зрозуміти її корені. Для тих, хто працює зі складними асинхронними системами або багатопотоковими додатками.

Yappi – це не інструмент. Це ваш помічник у створенні по-справжньому ефективного коду. Він не просто покаже цифри – він навчить вас бачити ваш код

по-новому. І коли ви звикнете до нього, ви вже не зможете уявити собі роботу без нього.

Спробуйте – і ви побачите свій код зовсім іншими очима.

1.8 Інструменти для виявлення "мертвого" та неефективного коду

Кожен розробник стикається з ситуацією, коли код перестає бути чистим і ефективним. З часом у проекті накопичуються функції, які ніхто не використовує, зайві імпорти або ділянки програми, що сповільнюють роботу. Ось як можна виявити такі проблеми без зайвих складнощів.

Шукаємо "мертвий" код

1. Автоматичний пошук невикористаних функцій

Деякі інструменти можуть просканувати проект і показати, які функції, змінні чи класи ніде не викликаються. Наприклад, можна запустити спеціальний скрипт або використати легкі бібліотеки, які аналізують код без зайвого навантаження.

2. Перевірка імпортів

Часто у файлах залишаються імпорти, які давно не використовуються. Вони не заважають роботі, але збільшують час запуску та заплутують код. Простий спосіб – видалити підсвічені IDE імпорти або скористатися інструментами, що знаходять зайві включення бібліотек.

3. Аналіз гілкування коду

Буває, що частина коду ніколи не виконується через умовні оператори (if/else), які завжди повертають одне й те саме значення. Можна протестувати різні сценарії або використати профілювальники, щоб побачити, які рядки програми взагалі не спрацьовують.

Виявляємо неефективність

4. Вимірювання часу виконання

Якщо функція працює повільно, це не завжди очевидно. Можна вручну заміряти час її роботи за допомогою time.time(), але зручніше – увімкнути

вбудований профілювальник, який покаже, які частини коду займають найбільше ресурсів.

5. Контроль пам'яті

Іноді код витрачає забагато оперативної пам'яті через неоптимальні структури даних або неочищені об'єкти. Існують бібліотеки, які допомагають відстежити, де і скільки пам'яті використовується.

6. Перевірка на зайві цикли та запити

Надмірні вкладені цикли, дублювання запитів до бази даних або API можуть значно гальмувати програму. Іноді досить переписати один блок, щоб прискорити роботу в рази.

Що робити з знайденими проблемами?

1) Видаляти без жалю – якщо код точно не використовується.
2) Оптимізувати – замість повільної функції знайти ефективніший спосіб.
3) Коментувати або документувати – якщо код поки що не потрібен, але може знадобитися пізніше.

Головне – не накопичувати "технічний борг". Краще час від часу перевіряти проект і прибирати зайве, ніж потім розбиратися з хаосом у тисячі рядків коду.

1.9 Огляд комплексних систем оптимізації коду

Оптимізація коду на Python давно перестала бути простою формальністю - це стратегічний процес, який впливає на всі аспекти розробки. Сьогодні існують цілі екосистеми інструментів, що дозволяють підняти якість коду на новий рівень.

Інтелектуальні помічники для розробників

Сучасні IDE трансформувалися з простих текстових редакторів у справжніх асистентів програміста. Вони не просто підсвічують синтаксичні помилки, а й пропонують конкретні шляхи оптимізації. Наприклад, можуть

запропонувати замінити класичний цикл на більш ефективний генератор, або переписати важку функцію з використанням вбудованих інструментів мови.

Ці системи аналізують код у реальному часі, враховуючи контекст усього проекту. Вони "розуміють", які частини програми вимагають особливої уваги, і можуть запропонувати рішення, адаптовані під конкретний стиль коду.

Хмарні аналітичні платформи

З'явився цілий клас сервісів, які проводять глибинний аналіз коду без необхідності встановлювати додаткове ПЗ. Достатньо підключити репозиторій - і система почне виявляти не лише очевидні проблеми, а й приховані неефективності.

Такі платформи особливо корисні для великих команд. Вони дозволяють відстежувати, як зміни в одній частині системи впливають на продуктивність в цілому. А візуалізація результатів у вигляді зрозумілих графіків робить аналіз доступним навіть для нетехнічних фахівців.

Системи моніторингу в реальному часі

Для серйозних проектів вже недостатньо оптимізувати код перед релізом. Потрібні інструменти, які постійно аналізують роботу програми в "бойових" умовах. Вони фіксують:

Як код поводить себе під навантаженням

Де виникають непотрібні затримки

Які компоненти споживають занадто багато ресурсів

Ці дані дозволяють проводити "точечні" оптимізації саме тих ділянок коду, які реально гальмують роботу системи.

Автоматизований рефакторинг

Найновіші інструменти вже не просто знаходять проблеми - вони можуть автоматично їх виправляти. За допомогою машинного навчання аналізуються мільйони рядків коду, і на основі цього досвіду генеруються конкретні пропозиції щодо вдосконалення.

Це особливо цінно для великих legacy-проектів, де ручний рефакторинг займав би місяці. Система може запропонувати цілий план оптимізації, враховуючи специфіку проекту і прийняті в команді стандарти коду.

Перспективи розвитку

Майбутнє оптимізації Python-коду - це інтеграція всіх цих підходів у єдиний безперервний процес. Від написання першого рядка коду до його виконання в production - кожен етап буде супроводжуватися інструментами аналізу та вдосконалення.

Головне - пам'ятати, що будь-яка оптимізація має бути обґрунтованою. Іноді простіше і ефективніше переписати ключові компоненти, ніж намагатися "вичавити" ще 5% швидкодії зі старого коду.

ВИСНОВКИ ПО РОЗДІЛУ 1

У першому розділі досліджено, як перетворити звичайний код на елегантне та продуктивне рішення. Оптимізація виявилася не просто технічним процесом, а справжнім мистецтвом балансу між швидкодією, читабельністю та простотою підтримки.

В цьому розділі з'ясовано, що гарний код - це не лише про швидкість виконання. Важливо, щоб він був:

1) Зрозумілим для колег через чітку структуру
2) Простим у вдосконаленні завдяки логічній організації
3) Ефективним у роботі за рахунок продуманих алгоритмів

Особливу увагу було приділено візуальній стороні програмування. Виявилося, що дотримання єдиного стилю - це не просто формальність. Коли код виглядає акуратно, з ним приємніше працювати, а помилки стають помітніші. Для цього існують спеціальні інструменти, які автоматично приводять код до ідеального стану.

Не менш цікавим став аналіз "хворих місць" програм. Ми навчилися знаходити:

1) Ділянки коду, які ніхто не використовує
2) Функції, що працюють повільно
3) Зайві операції, які сповільнюють роботу

Найцікавішим відкриттям стали комплексні системи, які поєднують усі ці підходи. Вони не просто знаходять проблеми, а й допомагають їх вирішувати, пропонуючи готові рішення для покращення коду. Це надихнуло мене на створення власного інструменту, який буде по-справжньому корисним для розробників.

У підсумку хочеться сказати, що оптимізація - це постійний процес вдосконалення. Код, як живий організм, потребує регулярного догляду та уваги. І тепер ми знаємо, як це робити ефективно.

РОЗДІЛ 2 ЗАДАННЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ОБРАНИЙ ІНСТРУМЕНТАРІЙ

2.1 Задання вимог до програмного забезпечення

Програмне забезпечення, розроблене в межах даного дослідження, повинно відповідати низці вимог, які охоплюють як функціональні, так і нефункціональні аспекти. Основною вимогою є реалізація системи мовою програмування Python із застосуванням бібліотеки PyQt5 для побудови графічного інтерфейсу. Архітектура програми повинна мати модульну структуру, де кожна його частина виконує окрему функцію: оптимізацію, аналіз або управління.

Інтерфейс користувача має містити поле для введення програмного коду, кнопку для запуску його обробки та вікно, у якому виводиться результат трансформації. Основна функціональність системи охоплює алгоритми стилістичної оптимізації, серед яких передбачено автоматичне форматування коду згідно зі стандартом PEP 8, додавання шаблонних коментарів до функцій і класів з описом вхідних та вихідних параметрів, а також переформатування і сортування імпортів. Окрему увагу приділено інтеграції з синтаксисом, подібним до мов C/C++ або Java: реалізовано механізми трансформації коду між блоковим стилем із фігурними дужками та стилем з відступами, характерним для Python. Для зручності також реалізовано зворотну трансформацію.

Крім того, програмне забезпечення реалізує алгоритми, спрямовані на підвищення продуктивності коду. Серед них — виявлення та видалення мертвого коду, оптимізація рекурсивних викликів шляхом додавання кешування, обгортання обчислювально складних функцій у високопродуктивні декоратори на кшталт numba.njit, а також розгортання простих циклів з передбачуваною кількістю ітерацій. Додатково реалізовано статичний аналіз коду, який дає змогу виявляти помилки синтаксису, логіки та стилю за допомогою модуля flake8.

Програмне забезпечення також містить модулі з налаштуванням користувацьких пресетів, що забезпечують гнучкість і можливість адаптації до

конкретних потреб розробника. Усі функції мають забезпечувати збереження логіки початкового коду, працювати з середніми обсягами даних із мінімальними затримками та надавати інтерфейс, зручний для взаємодії й доступний для локалізації.

2.2 Обрання мови та середовища програмування

Python — одна з найпопулярніших мов програмування, і це не випадково. Вона поєднує простоту, потужність та гнучкість, що робить її ідеальним вибором як для початківців, так і для досвідчених розробників. Ось декілька ключових переваг Python [8]:

1. Простота та читабельність

Python має інтуїтивно зрозумілий синтаксис, схожий на звичайну англійську мову. Це дозволяє швидко освоїти основи та писати код, який легко підтримувати. Наприклад, цикл for виглядає набагато лаконічніше, ніж у інших мовах.

2. Велика спільнота та багата бібліотека

Ця мова має величезну кількість бібліотек для різних завдань: від веброзробки (Django) до наукових досліджень (Pandas) та штучного інтелекту (TensorFlow). Це значно прискорює розробку, оскільки не потрібно писати код з нуля.

3. Крос-платформеність

Працює на всіх популярних операційних системах (Windows, macOS, Linux). Написаний код можна запускати різних середовищах без значних змін.

4. Висока продуктивність розробки

Завдяки динамічній типізації та автоматичному керуванню пам’яттю (зборник сміття) програміст може зосередитися на логіці, а не на технічних деталях. Це робить Python чудовим інструментом для швидкого прототипування.

5. Підтримка багатьох парадигм програмування

Python підтримує:

1) Процедурне програмування (функції, модулі)

2) ООП (класи, спадкування)

3) Функціональне програмування (лямбда-функції, генератори)

6. Застосування в перспективних галузях

Python лідирує в таких сферах, як:

1) Data Science (аналіз даних, машинне навчання)

2) Веброзробка (бекенд, API)

3) Автоматизація (скрипти, парсинг)

Робототехніка та IoT (бібліотеки для Raspberry Pi)

Висновок

Ця мова поєднує доступність для новачків із потужними інструментами для професіоналів. Її універсальність, велика спільнота та постійний розвиток роблять її однією з найкращих інструментів для навчання та комерційної розробки.

Розробка на Python може бути ще ефективнішою з правильним інструментом, і PyCharm від JetBrains — один із найкращих варіантів. Це не просто текстовий редактор, а повноцінне середовище розробки (IDE), яке значно прискорює написання коду, налагодження та тестування [7].

Чому він корисний для розробників?

1. Інтелектуальний редактор коду

Він розуміє Python на глибокому рівні:

1) Автодоповнення показує варіанти під час набору, враховуючи поточний контекст.

2) Підсвітка помилок до того, як код буде запущено.

3) Рефакторинг дозволяє легко перейменовувати змінні, класи та функції без ризику пошкодити логіку.

2. Вбудовані інструменти для продуктивності

1) Налагодження з візуалізацією кроків та змінних.

2) Віртуальні середовища (venv) — створення та керування без виходу з IDE.

3) Термінал та інтеграція з Git — можна робити коміти, пушити та мержити прямо в PyCharm.

3. Підтримка веб-технологій та фреймворків

Якщо ви працюєте з:

1) Django, Flask, FastAPI — PyCharm пропонує шаблони проектів, підказки для різних шляхів і шаблонів.

2) JavaScript/HTML/CSS — підтримка фронтенд-розробки в одному середовищі.

3) Бази даних — вбудований інструмент для роботи з різного типу базами данних.

4. Гнучкість та індивідуальні налаштування

1) Теми оформлення (Dark Mode — обов’язково!).

2) Плагіни (наприклад, для підтримки Docker, Kubernetes, Jupyter Notebook).

3) Інтеграція з іншими інструментами (наприклад, pytest, Black для автоформатування).

5. Версії PyCharm: Community vs Professional

1) Community (безкоштовна) — ідеальна для навчання та стандартних проєктів.

2) Professional (платна) — додає підтримку веб-фреймворків, профілювання коду, віддалену розробку тощо.

При розробці використано безкоштовну пробну версію Professional.

Висновок

PyCharm — це не просто "текстовий редактор з підсвіткою синтаксису", а потужний помічник Python-розробника. Він економить час, зменшує кількість помилок і робить процес кодингу комфортнішим.

2.3 Обрання бібліотек

Запропонована мною мова має багато корисних інструментів для роботи з кодом. Ось деякі з них, які варто знати.

Модуль ast дає змогу зазирнути під капот коду. Він перетворює програмний код на спеціальну деревоподібну структуру, з якою можна працювати як з об'єктом. Це корисно, коли потрібно проаналізувати або змінити код програмним шляхом. Наприклад, можна знайти всі виклики певної функції у великому проекті.

Для зворотного перетворення дерева у код є astor та astunparse. Вони беруть об'єкти синтаксичного дерева і генерують з них звичайний Python-код. Це дуже допомагає при написанні інструментів для автоматичного рефакторингу або кодогенерації.

Коли мова йде про стиль коду, на допомогу приходять black, isort та flake8. Black - це не просто форматер, а своєрідний "диктатор стилю", який примусово приводить код до єдиного стандарту. Isort сортує імпорти, групуючи їх за типами. Flake8 - це комбінований інструмент, який перевіряє код на відповідність PEP 8, знаходить логічні помилки та потенційні проблеми.

Для роботи з файловою системою та процесами є tempfile і subprocess. Tempfile створює тимчасові файли, які автоматично видаляються після використання. Subprocess дозволяє запускати зовнішні програми та керувати ними прямо з Python-коду.

Декоратор lru_cache з functools - це простий спосіб оптимізувати функції. Він запам'ятовує результати викликів і, коли функцію викликають з тими самими аргументами вдруге, повертає готовий результат замість повторного обчислення. Особливо корисно для рекурсивних функцій або складних обчислень.

Кожен з цих інструментів вирішує конкретні задачі. Разом вони утворюють потужний набір для професійної роботи з Python-кодом. Від аналізу та зміни коду до підтримки стилю та оптимізації - ці бібліотеки покривають

майже всі аспекти розробки. Головне - знати, коли і який інструмент застосувати.

ВИСНОВКИ ПО РОЗДІЛУ 2

У процесі розробки програмного забезпечення було визначено ключові вимоги до системи, що дозволило чітко сформулювати її функціонал та очікувані результати.

Вибір мови програмування та середовища розробки ґрунтувався на аналізі їх переваг, продуктивності та відповідності поставленим завданням. В якості основних інструментів було обрано PyQt5 та Pycharm.

Підібрано необхідні бібліотеки, які значно спрощують реалізацію окремих модулів та забезпечують надійність роботи програми.

Усі етапи підготовки до створення системи дозволили закласти міцну основу для подальшої розробки та впровадження проекту.

РОЗДІЛ 3 РОЗРОБКА СТРУКТУРИ ТА ІНТЕРФЕЙСУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Розробка структури

Опис структури

Почнемо з основного опису складу програми

Вона буде включати в себе тринадцять модулів (рис.3.1).

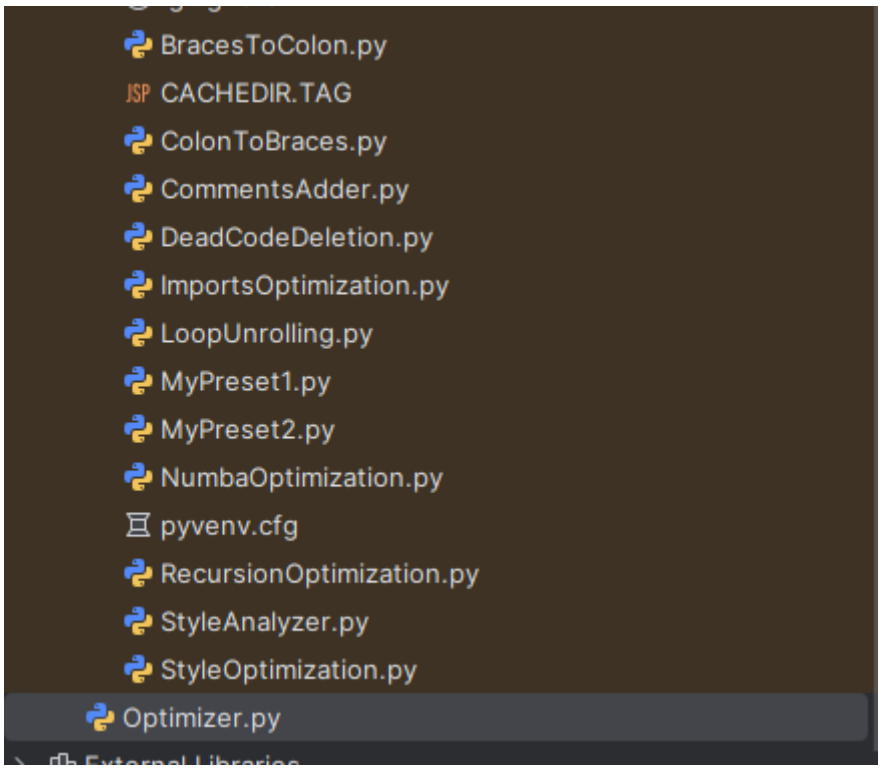


Рисунок 3.1 - Фрагмент потенційної структури програми відповідно до файлового менеджера

Основний модуль Optimizer - він складатиметься з функції запуску програми та класу TextProcessorApp, який як аргумент буде приймати віджет. Сам клас буде складатися з трьох функцій, ось відповідні задачі цих функцій:

1. Конструктор класу
2. Створення UI частини програмного продукту
3. Обробка введеного тексту ввімкненими алгоритмами

Необхідно додати, що функція завантаження додатку буде містити в собі стилізацію всієї системи.

Перейдемо до інших модулів. Кожен з залишившихся частин має в собі алгоритм або підключення бібліотеки з методом для оптимізації коду. Всього їх можна поділити на чотири категорії.

Перша категорія - оптимізація естетичної складової. Вона представлена за допомогою п’яти класів:

1. BracesToColon
2. ColonToBraces
3. StyleOptimization
4. CommentsAdder
5. ImportsOptimization

Всі алгоритми іменувалися відповідно до їх прямих задач.

BracesToColon - клас необхідний для перетворення обрамлення блочних конструкцій, а саме конвкертування фігурних дужок в двокрапку.

ColonToBraces – робить обернену до вище зазначеного класу дію, а саме трансформацію з двокрапок в фігурні дужки.

StyleOptimization – створений для приведення коду до єдиного стилю, реалізований з допомогою зовнішньої бібліотеки.

CommentsAdder – додає уточнюючі шаблони коментарів до класів та функцій.

ImportsOptimization – оптимізовує естетичну складову імпортів шляхом їх групування та зміни розташування.

Друга категорія –це алгоритми покращення швидкодії коду:

1) RecursionOptimization
2) DeadCodeDeletion
3) NumbaOptimization
4) UnrollingLoop

Аналогічно до першої категорії –методи іменуються відповідно до їх задач

RecursionOptimization – пошук та оптимізація рекурсій шляхом обрамлення в конструкції для кешування.

DeadCodeDeletion - видалення шматків коду які не використовуються або не викликаються в коді.

NumbaOptimization – пошук повільних шматків коду та обрамлення в необхідну для перетворення в машинний код бібліотеку.

UnrollingLoop – розгортання простих циклів.

До фінальної категорії можна віднести MyPreset1 та MyPreset2 – необхідні для створення разом з класом аналізатора

Підсумовуючи в цьому розділі спроектовано структуру програми (рис 3.2).

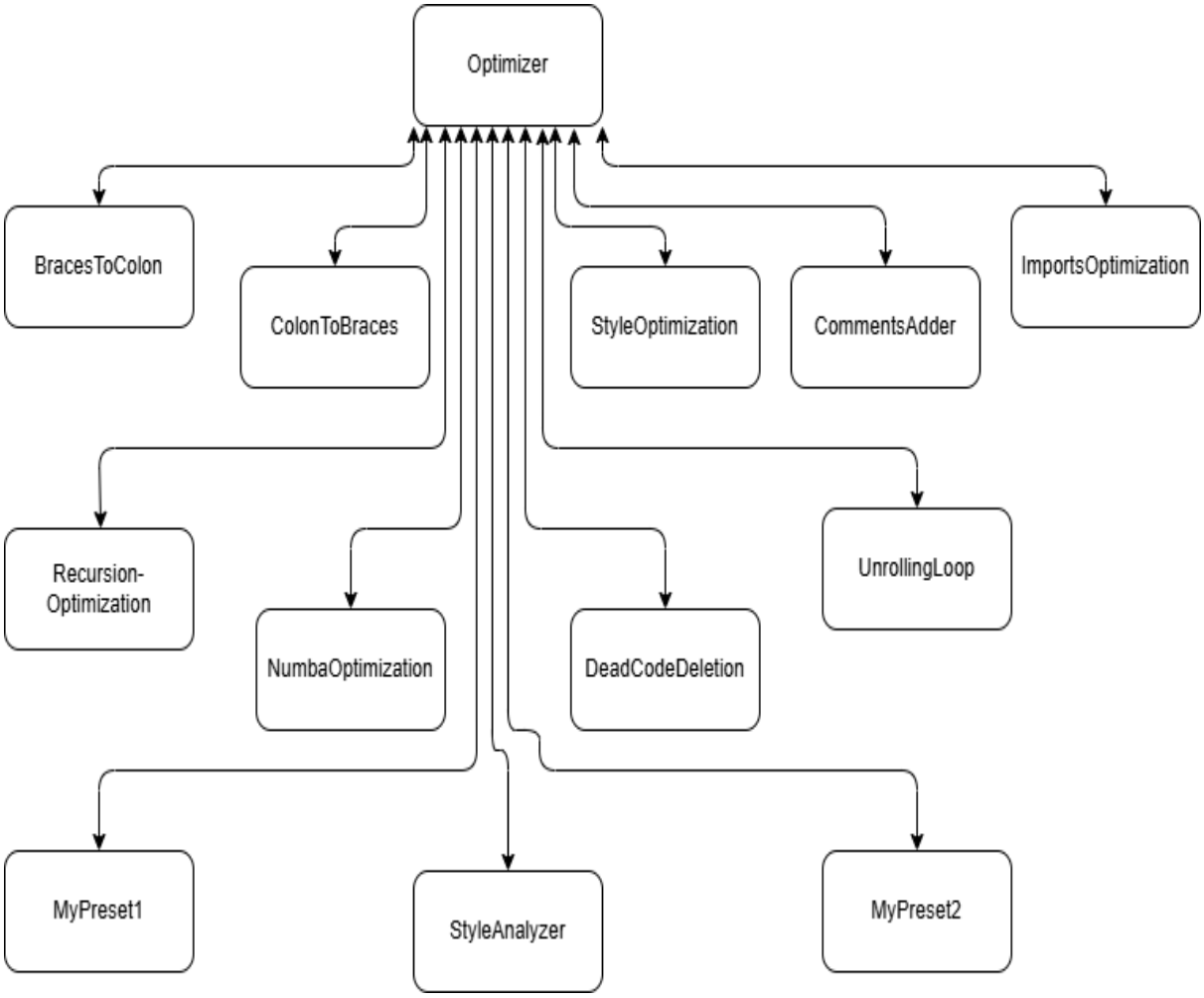


Рисунок 3.2- Взаємодія між модулями системи

Також було сформовано потенційні класи та взаємодії між ними необхідні для роботи програми.

3.2 Розробка інтерфейсу

Було створено спроектовано простий, але одночасно зручний інтерфейс (рис. 3.3)

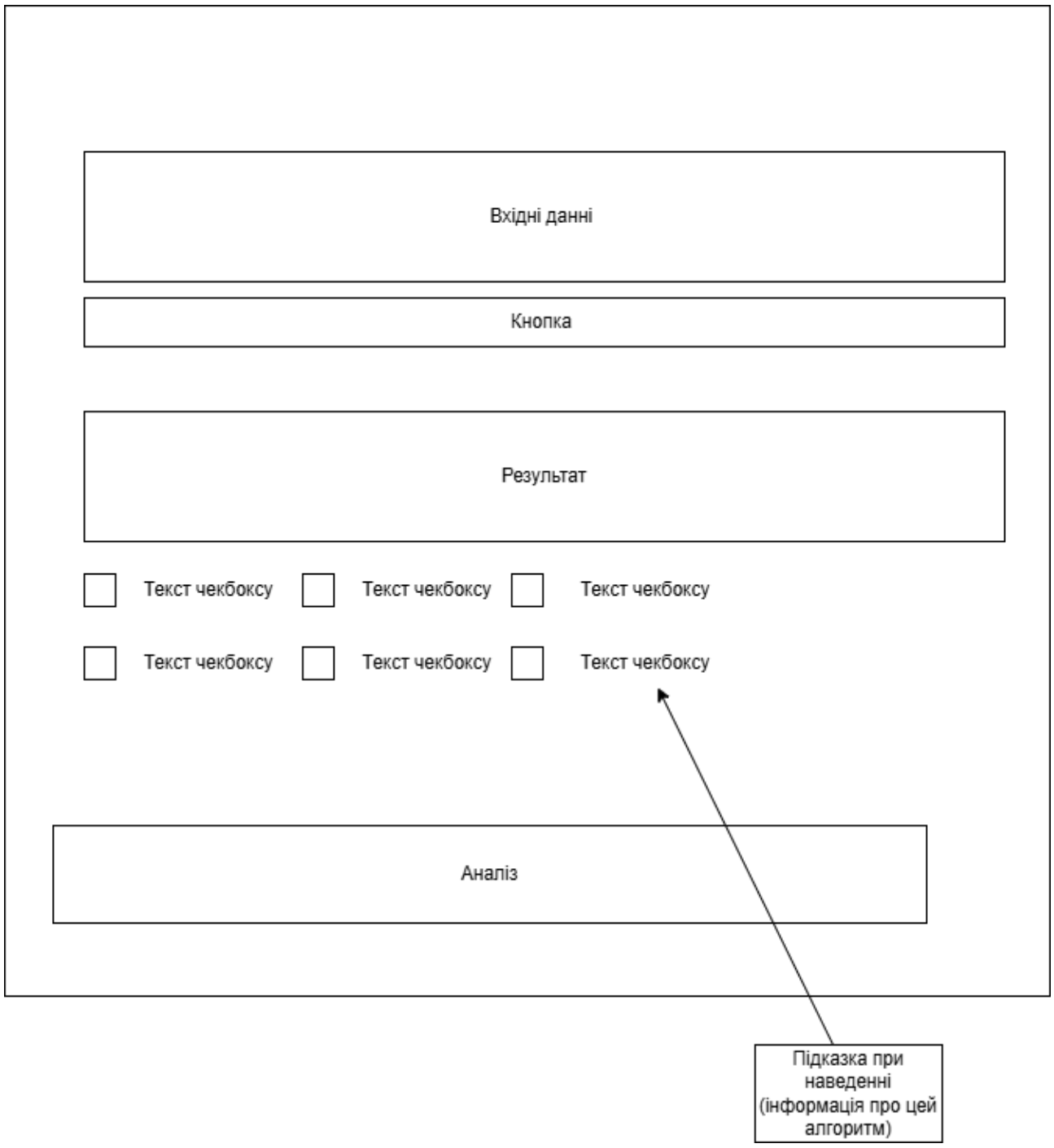


Рисунок 3.3 – Потенційний інтерфейс програмного забезпечення

Всього сам інтерфейс можна розбити на:

1. Вхідну форму разом з кнопкою
2. Виведення результату
3. Чекбокси котрі відповідають за вимкнення та ввімкнення аналізу або алгоритму оптимізації
4. Коротенький аналіз
5. Підказки до чекбоксів

Вхідна форма приймає в себе код або фрагмент коду необхідний для оптимізації. За допомогою натискання кнопки виводиться результат, який залежить від обраних чекбоксів методів. Та показується результат аналізу обробки коду.

Важливо додати, що чекбокси будуть згруповані відповідно до категорії.

Також необхідно зазначити наявність підказок до чекбоксів, котрі коротко пояснюють назву, функцію та конструкції, які будуть трансформовуватися. Інтерфейс зроблено простим і зручним для забезпечення біль

3.3 Підключення стилів

Стилізація додатку (рис. 3.4) буде додана в модулі управління системи та підключена за допомогою спеціальної функції PyQt бібліотеки. Адже як ми знаємо в сучасному світі дизайн – одна із найважливіших частин кожної системи. Основні кольори, які будуть використовуватися в системі:

1) Чорний
2) Синій
3) Білий

Чорний буде використаний для фону. Синій для демонстрації виділених чекбоксів. Білий стане важливою частиною всіх написів.

До речі про шрифт, обрано Segoe UI з різними розмірами в залежності від важливості лейблу.

Більше інформації про стилізацію можна отримати, якщо проаналізувати рисунок 3.4

```
QWidget {
    font-family: 'Segoe UI', sans-serif;
    font-size: 12pt;
    background-color: #1e1e1e;
}
QLabel {
    color: #e0e0e0;
    padding: 6px 0;
}
QTextEdit {
    background-color: #2d2d2d;
    border: 1px solid #3a3a3a;
    border-radius: 6px;
    padding: 8px;
    color: #ffffff;
    selection-background-color: #3a6ea5;
}
QPushButton {
    background-color: #3a3a3a;
    border: 1px solid #4a4a4a;
    border-radius: 6px;
    padding: 8px 16px;
    min-width: 100px;
    color: #e0e0e0;
}
QPushButton:hover {
    background-color: #4a4a4a;
}
QPushButton:pressed {
    background-color: #2a2a2a;
}
QCheckBox {
    spacing: 8px;
    color: #e0e0e0;
}
QCheckBox::indicator {
    width: 18px;
    height: 18px;
    border: 1px solid #5a5a5a;
    border-radius: 4px;
    background-color: #2d2d2d;
}
QCheckBox::indicator:checked {
    background-color: #3a6ea5;
    border: 1px solid #3a6ea5;
}
```

Рисунок 3.4 – Стилізація програми написана мовою CSS

ВИСНОВКИ ПО РОЗДІЛУ 3

У ході розробки структури програмного забезпечення було запропоновано оптимальну архітектуру, що забезпечує зручну організацію коду та масштабованість проекту.

Особливу увагу приділено створенню інтуїтивно зрозумілого інтерфейсу, який відповідає сучасним стандартам UX/UI та задовольняє потреби користувачів.

Для покращення візуальної складової програми підключено стилі, що дозволило досягти гармонійного дизайну та зручної взаємодії. Реалізація цих етапів заклала основу для подальшого функціоналу та успішного впровадження програмного продукту.

РОЗДІЛ 4 ОПИС ЗАПРОПОНОВАНОЇ СИСТЕМИ МЕТОДІВ ТА ТЕСТУВАННЯ

4.1 Опис комплексу методів

Опишем кожен алгоритм покроково.

Алгоритм приведення стилізації до загальноприйнятих норм

Цей метод пропонує форматування Python-коду з допомогою інструменту black. Він не даремно є одним із найпопулярніших бібліотек для приведення стилю. Нижче - опис створеного мною алгоритму на основі цього форматера:

Алгоритм роботи функції (рис. 4.1):

```
def optimize_code_with_black(self, code: str) -> str:
    try:
        # Mode configuration (можна змінити під себе)
        mode = black.FileMode()
        # Форматування коду
        formatted_code = black.format_str(code, mode=mode)
        return formatted_code
    except Exception as e:
        return f"Error formatting code: {e}"
```

Рисунок 4.1 - Алгоритм приведення стилізації до загальноприйнятих норм

Спочатку налаштовуємо режим шляхом створення кофігураційного об’єкта самої бібліотеки black, який вважається стандартним шляхом та визначає самі норми приведення коду. В цьому випадку ми використовуємо базовим – без модифікацій. Потенційно використовуючи цей конфігуратор ми маємо можливість зміни параметрів відступів, рядкової довжини, але оскільки я хочу зробити інтуїтивно просту програму та зручну в користуванні ми залишимо його без змін

Надалі бачимо як використовується функція format_str з двома аргументами:

1. Перший – це сам код для приведення
2. Конфігуратор котрий ми детально описали вище та залишили без змін

Сама функція повертає вже відформатований код.

Обробка вийнятків. Алгоритм знаходиться в блоці try-except, що надає нам можливість впіймати потенційні помилки, які можуть статися при обробці коду. При виниканні вийняток буде перехоплено та показано в консолі. Це забезпечує можливість в збереженні стабільності програми і описує потенційні проблеми і недоліки.

При успішній роботі функції повертається результат у вигляді вже оптимізованого коду, а при помилці ми побачимо в консолі повідомлення.

Підсумки по алгоритму

В підсумках хочеться описати переваги використаного інструменту, а саме:

1)Ідемпотентність

Повторний запуск алгоритму на вже оптимізованому ним коді покаже аналогічний результат, не змінюючи його, аналогічно до подібних функцій в математиці.

2) Інварітивність семантики

Поведінкова складова, а саме логіка коду – залишається незмінною після виконання цього прекрасного алгоритму.

3) Мінімалізація змін

Так зване зменшення «дифів», це непомітна якість, але цей інструмент в деяких випадках старається зменшити кількість змін необхідних для форматування коду до заданих стандартів.

Також опишемо самі механізми роботи цієї бібліотеки;

1) Оформлення відступів

Структурні частини коду мають строго однакову чисельність пробілів, а саме чотири штуки не більше – не менше.

2) Лапкова уніфікація

Всі наявні в коді лапки приводяться до подвійних («»), якщо це не порушує структуру рядка.

3) Обмеження довжини рядка

Код автоматично дробиться, якщо перевищує ліміт у 88 символів. Це не легше читати, також наслідок цього наявна краща сумісність з терміналами та редакторами.

4) Інтелектуальне форматування виразів

Бібліотека дробить комплексні вирази на логічні точки — наприклад, аргументи функцій розміщуються на окремих рядках з вирівнюванням.

5) Консервативне форматування

Якщо конструкція вже є коректною та компактною (наприклад, короткий словник або лямбда), Блек може залишити її незмінною, не створюючи зайвої вертикальної складності.

Також не треба забувати про сервісні функції наявні в класі (рис. 4.2)

```
def on_state_change(self, state):
    if state == Qt.Checked:
        self.button_state = True
    else:
        self.button_state = False

def process(self, text):
    if self.button_state:
        return self.optimize_code_with_black(text)
    return text
```

Рисунок 4.2 – Сервісні функції класу

Алгоритм додання коментарів

Алгоритм заснований на функції add_missing_docstrings, яка призначена для автоматизованого додавання базових докстрінгів, які мають вигляд формату """...""") до функцій і класів, за умови, що вони їх не мають. ЇЇ основну мету можна описати, як - зробити код більш зрозумілим з точки зору логіки

коду, при цьому допомогати дотримуватись принципів уточнення логіки функцій шляхом документування, притаманних Python. Створені структури у вигляді коментарів пояснюють призначення об'єктів, їхні аргументи та значення, які повертаються.

Сам аналіз починається з передавання рядка, який містить Python-код, до запропонованого методу у формі тексту. Потім ми працюємо з кодом як з древом структурним, а для цього ми використовуємо такий засіб як Abstract Syntax Tree. Це влаштована в мову бібліотека, котра перетворює програму в конструкцію типу дерева. Цей підхід допомагає визначити місце класів, функцій – звичайних та асинхронних та інших конструкцій.

Описуючи перший етап (рис 4.3) видно спроби перетворення в це дерево за допомогою «parse». Також варто пояснити, що завдяки обрамленню в конструкцію вийнятків, якщо заданий код буде з похибками, функція видасть помилку, яка буде перехоплюватись та відображатись з зручним описом для користувача.

```
try:
    tree = ast.parse(code)
except SyntaxError as e:
    raise ValueError(f"Невірний синтаксис Python: {e}")
```

Рисунок 4.3 – Перший етап алгоритму

Надалі бачимо створення внутрішнього класу з успадкуванням «NodeVisitor» (рис. 4.4).

```
class DocstringVisitor(ast.NodeVisitor):
    def __init__(self):
        self.nodes_to_document = []
```

Рисунок 4.4 – Внутрішній клас

Це звичайний підхід для обробки синтаксичного дерева, а саме створення «візитера», який здатний пересуватися вузлами його структури та робити певні обчислення або перевірки. Також додатково створимо та визначимо, як список – структуру в якій ми зберігатимемо класи та функції без уточнюючих коментарів.

На наступному етапі (рис. 4.5) бачимо функцію, котра перевіряє наявність коментаря в заданому відгалудженні. Відповідно до наявних стандартів, він має розміщуватися в якості першого компоненту в тілі структур функцій та класів. Це і є причиною навіщо ми будемо перевіряти принципом спочатку чи тіло пусте, потім на компонент як вираз, який є константою типу рядка. В разі відсутності цього об’єкту ми віднесемо його до недокументованого.

```
def _has_docstring(self, node):
    return (len(node.body) > 0 and
            isinstance(node.body[0], ast.Expr) and
            isinstance(node.body[0].value, ast.Constant) and
            isinstance(node.body[0].value.value, str))
```

Рисунок 4.5 – Функція перевірки наявності коментаря

Наступні методи (рис 4.6) мають на меті перебирання дерева з функціями – звичайними та асинхронними та класами і перевірки їх вище описаним методом. При знаходженні конструкції без коментаря вона додається до списку недокументованих. При чому додається в форматі «вузол – тип».

Після кінцевого формування списку, він сортується для забезпечення коректного додавання до результату. Після цього процесу відбувається подрібнення списку на рядки, при цьому не забуваючи про переходи, адже це запорука коректного додавання нових рядків.

Потім алгоритм пересувається елементами з браком документації та з наперед визначеним кількістю відступів, додається шаблон.

Формат шаблону:

1) Опис призначення структури
2) Категорія зі списком параметрів

3) Категорія з значеннями котрі повертаються

```
    def visit_FunctionDef(self, node):
        if not self._has_docstring(node):
            self.nodes_to_document.append((node, "function"))
        self.generic_visit(node)

    def visit_AsyncFunctionDef(self, node):
        if not self._has_docstring(node):
            self.nodes_to_document.append((node, "async function"))
        self.generic_visit(node)

    def visit_ClassDef(self, node):
        if not self._has_docstring(node):
            self.nodes_to_document.append((node, "class"))
        self.generic_visit(node)

visitor = DocstringVisitor()
visitor.visit(tree)
```

Рисунок 4.6 – Функції проходу по дереву

Якщо говорити про класи то в цьому випадку створюється та додається шаблон типу:

1) Опис призначення структури
2) Список атрибутів та потенційний їх опис

Шаблони котрі сформувалися додаються перед визначеннями пов’язаних конструкцій. Потім можна побачити трансформацію за якої все об’єднується в повний код та повертається в якості результату.

Не забуваємо про сервісні структури у вигляді функцій (рис. 4.7)

Підсумуємо, що незважаючи на свою просту складову цей метод ефективно та автоматично додає шаблони документації до заданих структур, роблячи ці шматки коду більш зрозумілими для розробників.

Алгоритм перетворення двокрапки в фігурні дужки

Давайте покроково розберемо кожний частину алгоритму. Перша частина (рис. 4.8) складається зі створення функції convert_python_to_braces, яка приймає рядок з Python-кодом, lines = code.splitlines() — розбиття коду на окремі рядки,

```
def on_state_change(self, state):
    if state == Qt.Checked:
        self.button_state = True
    else:
        self.button_state = False

def process(self, text):
    if self.button_state:
        return self.add_missing_docstrings(text)
    return text
```

Рисунок 4.7 – Сервісні структури алгоритму

ініціалізації списку «converted» для трансформованого коду та «indent_stack» — стек в якому я буду зберігати рівень відступу. Починається з нуля - перший рівень.

```
    def convert_python_to_braces(self, code: str) -> str:
        lines = code.splitlines()
        converted = []
        indent_stack = [0]
```

Рисунок 4.8 - Фрагмент ініціалізації алгоритму

Потім іде функція, яка допомагає визначити попередній рівень відступу, вона необхідні для зручного порівняння попереднього та поточних відступів (рис. 4.9)

```
def current_indent():
    return indent_stack[-1]
```

Рисунок 4.9 - Допоміжна функція алгоритму

В наступній частині (рис. 4.10) іде вже покрокове перебирання кожної строки коду, разом з аналізом та трансформацією, але спочатку ми перевіряємо чи є рядок пустим з а допомогою функції lstrip, якщо рядок - порожній, то переходимо на наступну ітерацію, якщо ні, то плавно дивимось наступну частину коду.

```
for i, line in enumerate(lines):
    stripped = line.lstrip()
    if not stripped:
        converted.append('')
        continue
```

Рисунок 4.10 – Фрагмент алгоритму

Далі дуже велика по важливості та об’єму частка алгоритму (рис. 4.11). Спочатку ми вимірюємо відступ рядка. І якщо бачимо, що він зменшився, то видаляємо останній рівень відступу зі стеку і додаємо } з відповідним відступом — закриваючи блок. В інакшому випадку редагуємо попередній рядок і замінюємо двокрапку на фігурну відкриваючу дужку, якщо вона там присутня, та протилежно до попереднього варіанту додаємо до стеку відступів поточний. В кінці проходу рядка додаємо конвертовану версію до попередньо зазначеного converted.

```
indent = len(line) - len(stripped)
while indent < current_indent():
    indent_stack.pop()
    converted.append(' ' * current_indent() + '}')

if indent > current_indent():
    indent_stack.append(indent)
    # Замінюємо ":" на " {"
    if converted:
        prev_line = converted.pop()
        if prev_line.rstrip().endswith(':'):
            prev_line = prev_line.rstrip()[:-1] + ' {'
        converted.append(prev_line)
# Звичайна обробка рядк
converted.append(' ' * indent + stripped)
```

Рисунок 4.11 - Основний фрагмент алгоритму

Також останній шматок алгоритму (рис. 4.12) закриває всі блоки, які залишились відкритими і працює за аналогічним принципом - видалення із стеку відступів та додавання до коду фігурної дужки.

```
# Закриваємо всі відкриті блоки
while len(indent_stack) > 1:
    indent_stack.pop()
    converted.append(' ' * current_indent() + '}')
return '\n'.join(converted)
```

Рисунок 4.12 - Закриваючий фрагмент алгоритму

Варто зазначити, що аналогічно до інших в класі цього алгоритму є дві сервісні функції process та on_state_change (рис. 4.13).

```
def on_state_change(self, state):
    if state == Qt.Checked:
        self.button_state = True
    else:
        self.button_state = False

def process(self, text):
    if self.button_state:
        return self.convert_python_to_braces(text)
    return text
```

Рисунок 4.13 - Сервісні функції

Обернена версія алгоритму

Початкова частина (рис. 4.14) складається з оголошення функції convert_braces_to_python одним із аргументів якої є код, розбивки на рядки за допомогою splitlines() функції та запису результату до lines, далі бачимо вже знайомий нам converted в який ми будемо записувати результати перетворень,

також новою особливістю порівняно з алгоритмом, який ми записували вище є заміна стеку відступів на їх рівень та кількість ми побачимо причини такої кардинальної зміни в майбутніх шматках алгоритму.

```
def convert_braces_to_python(self, code: str) -> str:
    lines = code.splitlines()
    converted = []
    indent_level = 0
    indent_size = 4  # Кількість пробілів на один рівень
```

Рисунок 4.14 - Ініціалізуючий фрагмент алгоритму

На наступному шматку (рис. 4.15) алгоритму можна побачити, що я створив майже аналогічну до попереднього алгоритму перевірку на пусті рядки. Якщо він чимось заповнений, то переходимо до наступної частини циклу.

```
for line in lines:
    stripped = line.strip()

    if not stripped:
        converted.append('')
        continue
```

Рисунок 4.15 - Шматок циклу алгоритму

У цьому блоці (рис. 4.16) створено список parts, який буде збірігати окремі фрагменти поточного рядка. Спочатку створюється порожній список parts та тимчасовий рядок temp, у якому ми поступово накопичуємо символи. Потім ми перебираємо кожен символ у рядку stripped, тобто у вхідному рядку без зовнішніх пробілів. Якщо створений алгоритм натрапляє на символ }, це означає завершення одного рівня вкладеності, тому я спершу додаю до parts усе, що накопичилось у temp, якщо воно не порожнє (це код перед дужкою). Потім саму дужку } я також додаю до parts як окремий елемент, адже вона виконує важливу

роль — зменшує рівень відступу. Після цього temp очищується для накопичення наступного фрагмента. Якщо символ не є }, то ми просто додаємо його до temp, поступово формуючи текстовий фрагмент.

Після завершення проходу по рядку, якщо в temp ще залишився код, ми додаємо його в parts як останній логічний елемент. Такий підхід гарантує, що код і дужки } будуть чітко розділені, навіть якщо вони трапляються в одному рядку вперемiш.

Також цей блок обробляє список parts, отриманий у попередньому кроці. Проходить по кожному елементу part і визначає, що з ним робити. Якщо це '}``, тобто ознака завершення вкладеного блоку, я зменшую рівень відступу indent_level, але не даю йому впасти нижче нуля, використовуючи max.

Це важливо, щоб уникнути негативних значень відступу. Якщо елемент закінчується на '{', це вказує на початок нового блоку. У цьому випадку робиться заміна дужки '{'на двокрапку':', формуючи правильний синтаксис Python. Потім додається рядок до converted незабуваючи про застосування відповідної кількості пробілів (відповідно до indent_level * indent_size`) і одразу збільшується рівень відступу, бо далі буде вкладений код.

Якщо фрагмент не є ані закриваючою дужкою, ані відкриваючою, це означає звичайний рядок коду. В алгоритмі я його додаю до результату з поточним рівнем відступу. Такий підхід дозволяє обробляти навіть ті рядки, які містять кілька блокових дужок разом з кодом, забезпечуючи правильний відступ для кожного фрагмента, відповідно до логіки вкладеності.

```
    parts = []
    temp = ''
    for char in stripped:
        if char == '}':
            if temp.strip():
                parts.append(temp.strip())
            parts.append('}')  # Позначаємо місце зменшення рівня
            temp = ''
        else:
            temp += char
    if temp.strip():
        parts.append(temp.strip())

    for part in parts:
        if part == '}':
            indent_level = max(indent_level - 1, 0)
        elif part.endswith('{'):
            line = part[:-1].rstrip() + ':'
            converted.append(' ' * (indent_level * indent_size) + line)
            indent_level += 1
        else:
            converted.append(' ' * (indent_level * indent_size) + part)

return '\n'.join(converted)
```

Рисунок 4.16 - Основна частина логіки

Не варто забувати і про сервісні функції які є невід’ємною частиною цього класу (рис. 4.17)

```
    def on_state_change(self, state):
        if state == Qt.Checked:
            self.button_state = True
        else:
            self.button_state = False

    def process(self, text):
        if self.button_state:
            return self.convert_braces_to_python(text)
        return text
```

Рисунок 4.17 – Сервісні функції

Алгоритм впорядкування імпортів

Для впорядкування імпортів у Python-програмах часто використовують спеціальні інструменти. Один із них — це бібліотека isort, яка автоматично наводить лад у списку імпортів. Розглянемо, як це реалізується на створеній мною функції (рис. 4.18).

```
def format_code_with_isort_and_black(self, code: str) -> str:
    #Оптимізація імпортів за допомогою isort
    code_with_sorted_imports = isort.code(code, profile="black")

    return code_with_sorted_imports
```

Рисунок 4.18 – Частина класу алгоритму

Що таке може ця бібліотека і навіщо вона потрібна?

Не забуваємо, що це зручна утиліта, яка допомагає автоматично сортувати імпорти у вашому коді. Вона не просто розставляє їх у алфавітному порядку, а й ділить на логічні групи: імпорти зі стандартної бібліотеки, зовнішні залежності та локальні модулі. Це робить код акуратнішим і легшим для читання.

До того ж вона добре "дружить" з іншими інструментами на кшталт black, що відповідає за загальний стиль форматування коду. Завдяки цьому імпорти оформлюються узгоджено з усім іншим кодом, з дотриманням відступів та порожніх рядків між групами.

Як працює функція впорядкування? Алгоритм дії доволі простий:

Отримання коду:

Спершу функція приймає Python-код у вигляді рядка. Важливо, щоб цей код уже містив імпортовані модулі, які потребують впорядкування.

Використання isort:

Далі код передається в isort.code(), із зазначеним параметром profile="black". Це означає, що сортування буде виконане з урахуванням правил форматування, прийнятих у black. isort проходить по всьому коду, знаходить

імпорти і сортує їх у правильному порядку, вставляючи необхідні розділення між групами.

Рузультат:

На виході можна побачити трансформований код, у якому всі імпорти впорядковані згідно з найкращими практиками. І у результаті отримуємо чистий, структурований код, готовий до подальшого використання або передавання іншим розробникам.

Навіщо це потрібно?

Чітко впорядковані імпорти — це не тільки естетична краса. А й підвищення таких характеристик як читабельність, і як результат полегшує підтримку проекту та знижує ризик конфліктів при об’єднанні змін у спільній роботі над програмним забезпеченням.

Алгоритм аналізу

Цей алгоритм потрібен для того, щоб автоматизовано перевірити Python-код на неточності стилю та синтаксису. Вона використовує інструмент flake8, який шукає проблеми у коді ще до початку виконання програми. Розберімося, що саме цей алгоритм (рис. 4.19) робить.

```
def analyze_code_with_flake8(self, code: str) -> str:
    """
    Аналізує Python-код за допомогою flake8 і повертає звіт про помилки.

    :param code: Рядок з Python-кодом.
    :return: Результат перевірки flake8 у вигляді тексту.
    """
    with tempfile.NamedTemporaryFile(mode='w+', suffix='.py', delete=False) as temp_file:
        temp_file.write(code)
        temp_file.flush()

        result = subprocess.run(
            args: ['flake8', temp_file.name],
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True
        )

    return result.stdout if result.stdout else "Немає помилок Flake8."
```

Рисунок 4.19 - Алгоритм аналізу

Крок 1: Тимчасовий файл

Спочатку функція створює тимчасовий .py файл — це просто місце, куди вона записує ваш код. Для цього використовується спеціальний модуль tempfile, який дозволяє працювати з файлами, що автоматично зникають після використання. Код записується у цей файл, і лише після цього переходить до аналізу.

Крок 2: Перевірка коду flake8

Після створення тимчасового файлу функція запускає flake8. Вона ніби каже йому: «Ось тобі файл, подивись, чи все в порядку». Це робиться через модуль subprocess, який дозволяє запускати зовнішні програми.

Вивід flake8 (усі помилки та попередження) збирається в змінну. Якщо flake8 знаходить щось неправильне — ці повідомлення будуть доступні для подальшого аналізу.

Крок 3: Результати

Якщо flake8 знаходить порушення — функція повертає список цих помилок. Якщо все добре — ви побачите повідомлення "Немає помилок Flake8".

Для чого це потрібно?

Це зручно, коли ви хочете швидко перевірити шматок коду без створення окремого файлу. Такий підхід особливо корисний у редакторах коду, середовищах для тестування або при перевірці чужих фрагментів коду.

Алгоритм оптимізації рекурсії

Представлений алгоритм реалізує автоматичну оптимізацію Python-коду шляхом виявлення рекурсивних функцій та застосування до них механізму кешування результатів. Функціональність реалізована через аналіз абстрактного синтаксичного дерева з подальшою трансформацією вихідного коду.

Основні етапи роботи алгоритму:

На етапі синтаксичного аналізу вихідний код трансформується в абстрактне синтаксичне дерево за допомогою модуля ast. При виявленні синтаксичних помилок у вихідному коді система генерує відповідне повідомлення про помилку без подальшої обробки.

Для виявлення рекурсивних конструкцій реалізовано спеціалізований візитор (рис. 4.20), який успадковує функціонал ast.NodeVisitor. Даний візитор здійснює обхід синтаксичного дерева з аналізом структури функцій та їх взаємних викликів. Виявлення рекурсії базується на аналізі збігів імен функцій між точками визначення та виклику.

```
class RecursionDetector(ast.NodeVisitor):
    def __init__(self):
        self.recursive_funcs = set()

    def visit_FunctionDef(self, node):
        self.current_func = node.name
        self.recursive_call_found = False
        self.generic_visit(node)
        if self.recursive_call_found:
            self.recursive_funcs.add(node.name)

    def visit_Call(self, node):
        if isinstance(node.func, ast.Name) and node.func.id == self.current_func:
            self.recursive_call_found = True
        self.generic_visit(node)

detector = RecursionDetector()
detector.visit(tree)
```

Рис. 4.20 – Детектор рекурсії

Після завершення аналізу система (рис. 4.21) перевіряє наявність виявлених рекурсивних конструкцій. При відсутності рекурсивних викликів вихідний код повертається без змін. У разі виявлення рекурсивних функцій алгоритм здійснює їх модифікацію.

Оптимізація коду полягає у додаванні декоратора @lru_cache перед визначенням кожної рекурсивної функції. Система автоматично додає необхідний імпорт з бібліотеки functools при першому використанні декоратора. Трансформація коду виконується зі збереженням вихідної структури та відступів.

Результатом роботи алгоритму є оптимізована версія вихідного коду, в якій усі рекурсивні функції захищені від надмірних обчислень за рахунок механізму мемоїзації. Ефективність такого підходу особливо помітна при обробці функцій

```
    if not detector.recursive_funcs:
        return code_str  # No recursion found

    lines = code_str.strip().split("\n")

    optimized_lines = []
    decorator_added = False

    for i, line in enumerate(lines):
        stripped = line.strip()
        if stripped.startswith("def ") and any(f"def {name}(" in stripped for name in detector.recursive_funcs):
            if not decorator_added:
                optimized_lines.append("from functools import lru_cache")
                decorator_added = True
            indent = line[:len(line) - len(line.lstrip())]
            optimized_lines.append(f"{indent}@lru_cache(maxsize=None)")
        optimized_lines.append(line)

    return "\n".join(optimized_lines)

except Exception as e:
    return f"# Error during optimization: {e}"
```

Рисунок 4.21– Друга частина алгоритму оптимізації рекурсії

з повторюваними вхідними параметрами, де кешування проміжних результатів дозволяє уникнути зайвих обчислень.

Алгоритм прискорення повільного коду

Представлений алгоритм здійснює автоматичну оптимізацію продуктивності Python-коду шляхом виявлення обчислювально складних функцій та їх подальшої трансформації з використанням JIT-компіляції. Основним інструментом оптимізації виступає декоратор @njit з бібліотеки Numba, який забезпечує компіляцію Python-коду у машинний.

Механізм роботи алгоритму ґрунтується на аналізі абстрактного синтаксичного дерева з подальшою кількісною оцінкою обчислювальної складності. Спеціалізований візитор, що успадковує функціонал ast.NodeVisitor, здійснює обхід синтаксичної структури, присвоюючи кожній функції ваговий коефіцієнт на основі наявних обчислювальних конструкцій.

Критерії оцінки складності включають аналіз циклічних конструкцій, математичних операцій та вкладених викликів функцій. Циклічні конструкції отримують підвищений ваговий коефіцієнт через їх високу обчислювальну

вартість. Прості арифметичні операції оцінюються значно нижче через їх оптимізовану реалізацію у інтерпретаторі Python (рис. 4.22).

```
class ComputeHeavyFunctionFinder(ast.NodeVisitor):
    def __init__(self):
        self.function_scores = {}

    def visit_FunctionDef(self, node):
        score = 0
        for subnode in ast.walk(node):
            if isinstance(subnode, (ast.For, ast.While)):
                score += 2
            elif isinstance(subnode, (ast.BinOp, ast.Call)):
                score += 1
        self.function_scores[node.name] = (score, node)
```

Рисунок 4.22– Клас для обчислення складності

Після завершення аналізу система виконує ранжування функцій за спаданням обчислювальної складності. Для найбільш ресурсномістких функцій автоматично додається декоратор JIT-компіляції. Алгоритм передбачає обов'язкову перевірку наявності необхідних імпортів та їх автоматичне додавання у разі потреби.

Трансформація коду здійснюється на рівні абстрактного синтаксичного дерева з подальшим відновленням текстового представлення. Такий підхід гарантує збереження вихідної структури коду та усіх коментарів. Результатом роботи алгоритму є оптимізована версія вихідного коду, де обчислювально складні функції автоматично адаптовані для виконання з максимальною ефективністю.

Ефективність запропонованого методу особливо помітна при обробці чисельних алгоритмів та наукових обчислень, де JIT-компіляція дозволяє досягти прискорення виконання на порядки. Алгоритм особливо корисний у сценаріях, де ручна оптимізація коду є трудомісткою або неможливою через великий обсяг кодової бази.

Алгоритм видалення «мертвого коду»

Представлений алгоритм реалізує систематичний підхід до виявлення та видалення нефункціональних елементів коду на основі аналізу абстрактного синтаксичного дерева. Основна мета полягає у підвищенні якості програмного коду через усунення марних конструкцій без втрати функціональності.

Механізм роботи алгоритму ґрунтується на двох основних компонентах: аналізаторі використання (рис. 4.23) та трансформаторі коду (рис. 4.24). Аналізатор використання, реалізований як спеціалізований візитор AST, здійснює комплексний обхід синтаксичної структури з метою фіксації всіх визначень та викликів програмних сутностей.

```python
class UsageAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.used_names = set()
        self.defined_functions = set()
        self.defined_classes = set()
        self.defined_variables = set()
        self.class_methods = {}

    def visit_Name(self, node):
        if isinstance(node.ctx, ast.Load):
            self.used_names.add(node.id)
        self.generic_visit(node)

    def visit_Call(self, node):
        if isinstance(node.func, ast.Name):
            self.used_names.add(node.func.id)
        elif isinstance(node.func, ast.Attribute):
            self.used_names.add(node.func.attr)
        self.generic_visit(node)

    def visit_FunctionDef(self, node):
        self.defined_functions.add(node.name)
        self.generic_visit(node)

    def visit_ClassDef(self, node):
        self.defined_classes.add(node.name)
        self.class_methods[node.name] = [
            n.name for n in node.body if isinstance(n, ast.FunctionDef)
        ]
        self.generic_visit(node)

    def visit_Assign(self, node):
        for target in node.targets:
            if isinstance(target, ast.Name):
                self.defined_variables.add(target.id)
        self.generic_visit(node)
```

Рисунок 4.23 – Аналізатор використання

Принцип роботи аналізатора передбачає реєстрацію:

1) Ідентифікаторів змінних при їх оголошенні та використанні

2) Функціональних визначень та їх викликів

3) Класових структур з аналізом використання методів

Трансформатор коду (рис. 4.24), що успадковує функціонал ast.NodeTransformer, виконує модифікацію синтаксичного дерева на основі даних, зібраних аналізатором.

```
class CodeCleaner(ast.NodeTransformer):
    def __init__(self, analyzer: UsageAnalyzer):
        self.analyzer = analyzer

    def visit_FunctionDef(self, node):
        if node.name not in self.analyzer.used_names:
            return None
        return self.generic_visit(node)

    def visit_Assign(self, node):
        if isinstance(node.targets[0], ast.Name):
            var_name = node.targets[0].id
            if var_name not in self.analyzer.used_names:
                return None
        return self.generic_visit(node)

    def visit_ClassDef(self, node):
        if node.name not in self.analyzer.used_names and not any(
                method in self.analyzer.used_names
                for method in self.analyzer.class_methods[node.name]
        ):
            return None
```

Рисунок 4.24 – Видалення структур

Основні правила трансформації включають:

1)Видалення нефункціональних функціональних визначень

2) Видалення класів об’єкти котрих не створюються в коді

Процес оптимізації здійснюється у строгій послідовності:

1) Побудова абстрактного синтаксичного дерева вхідного коду

2) Аналіз використання програмних сутностей

3) Трансформація дерева з видаленням нефункціональних елементів

4) Відновлення синтаксичної цілісності модифікованого коду

5) Генерація оптимізованого текстового представлення

Ефективність запропонованого методу особливо проявляється при обробці великих кодових баз, де ручний аналіз вимагає значних тимчасових витрат. Алгоритм забезпечує збереження семантичної еквівалентності коду після оптимізації, усуваючи лише ті елементи, які не впливають на виконання програми.

Важливою особливістю методу є збереження структурної цілісності коду після трансформації, що досягається за рахунок використання стандартних механізмів роботи з абстрактним синтаксичним деревом. Це забезпечує коректність результуючого коду та його відповідність вихідній семантиці.

Алгоритм розгортки циклів

Представлений алгоритм реалізує трансформацію ітеративних конструкцій мови Python у лінійну послідовність інструкцій шляхом статичного розгортання циклів. Метод ґрунтується на синтаксичному аналізі та перетворенні коду зі збереженням семантичної еквівалентності.

Основний механізм роботи передбачає два етапи аналізу:

Лексичний аналіз виконує ідентифікацію строкових літералів для виключення їх із подальшої обробки. Це забезпечує коректність трансформації, запобігаючи модифікації текстових фрагментів, які можуть містити схожі до циклів конструкції.

Синтаксичний аналіз здійснює розпізнавання циклічних структур за допомогою спеціалізованих шаблонів (рис 4.25):

Для for-циклів аналізується конструкція range() з визначенням параметрів ітерації

Для while-циклів виявляється ініціалізація змінної, умова продовження та крок інкременту

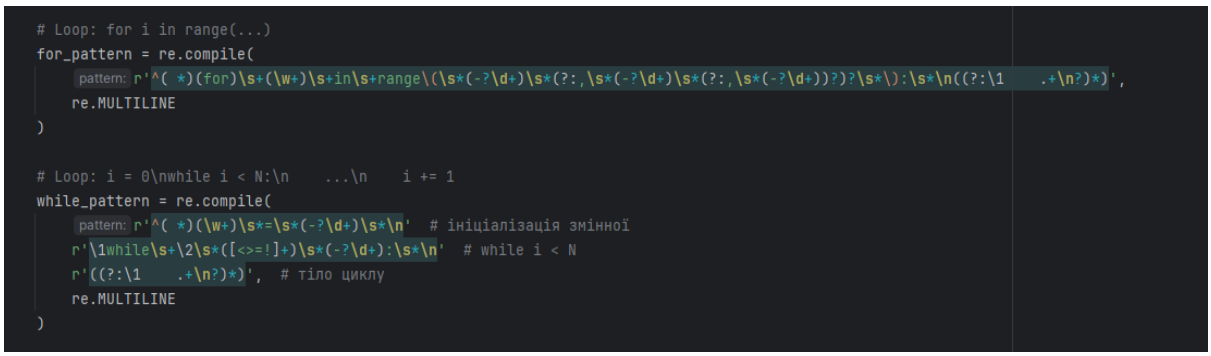


Рисунок 4.25 – Спеціалізовані шаблони

Трансформаційний етап алгоритму включає:

1. Генерацію послідовності значень ітератора на основі виявлених параметрів циклу

2. Створення копій тіла циклу для кожного значення ітератора

3. Підстановку конкретних значень замість змінної ітерації

4. Виключення інструкцій зміни стану ітератора

Особливістю методу є збереження структури вихідного коду та відступів, що забезпечує читабельність результуючого коду. Алгоритм демонструє особливу ефективність при обробці циклів з невеликою кількістю ітерацій, де витрати на дублювання коду компенсуються підвищенням продуктивності.

Результатом роботи алгоритму є лінійна послідовність інструкцій, яка семантично еквівалентна вихідним циклічним конструкціям, але:

1. Позбавляється накладних витрат на управління циклом

2. Дозволяє точніший аналіз потоків даних

3. Надає можливість подальших локальних оптимізацій

Запропонований метод особливо корисний у сценаріях, де потрібна статична аналітика коду або експерименти з продуктивністю, а також може бути використаний як попередній етап для інших видів оптимізації коду.

4.2 Заміри алгоритмів

Почнемо з замірів (таблиця 4.1) оптимізації естетичної складової

Заміри наскільки відсотків оптимізує той чи інший алгоритм проводилися за допомогою метрика, яка вираховувала абсолютну суму різниць кількості кожного символу в початковому коді та в результаті. До речі цю функцію вимірювання було додано до основної програми і реалізовано за допомогою спеціального створеного власноруч алгоритму (рис. 4.26).

Якщо подивитись на таблицю 4.1, то побачимо, що найбільш ефективною є оптимізація за допомогою всіх алгоритмів одночасно. На другому місці документування (додавання коментарів) в середньому покращення на ~36.562%, такий гарний результат пов’язаний з відсутністю уточнюючих коментарів в обраних мною шматках коду. На третьому місці black форматування з вражаючими середніми показниками в ~8,498%, далі йде

Фрагмент коду	Black	Додавання коментарів	Оптимізація імпортів	Трансформація стилізації блочних функцій	Всі разом
Взятий з Django REST Framework Tutorial	~3.48%	~47.95%	~0.12%	~3.98%	~54.66%
Взятий з Flask Official Examples	~10.38%	~24.32%	~0.55%	~12.02%	~45.90%
Взятий з Kaggle Notebook	~12.57%	~93.63%	~0.18%	~7.26%	~112.39%
Взятий з Django CookieCutter Template	~4.05%	~10.19%	~0.35%	~5.21%	~19.21%
Взятий з Flask RESTful Example	~12.01%	~6.72%	~0.26%	~5.30%	~24.94%

Таблиця 4.1– Таблиця різниць коду до і після оптимізації

трансформація стилізації блочних конструкцій та на останньому місці оптимізація імпортів, це пов’язано з тим, що зазвичай імпорти займають лише невеличку частину коду.

Також важливо додати неймовірні показники алгоритму додавання коментарів можна пояснити великою кількістю класів та функцій, які малі за об’ємом, але не мають документування. При збільшенні самого обсягу таких структур буде відбуватися пропорційне зменшення різниці оптимізації.

```
def analyze_texts(self, text1, text2):  1 usage
    # Функція для підрахунку кількості кожного символу в тексті
    def count_chars(text):
        char_count = {}
        for char in text:
            char_count[char] = char_count.get(char, 0) + 1
        return char_count

    # Підрахунок символів для обох текстів
    count1 = count_chars(text1)
    count2 = count_chars(text2)

    # Загальна кількість символів у першому тексті
    total_chars = len(text1)

    # Обчислення різниці між кількістю символів
    difference = 0
    all_chars = set(count1.keys()).union(set(count2.keys()))

    for char in all_chars:
        cnt1 = count1.get(char, 0)
        cnt2 = count2.get(char, 0)
        difference += abs(cnt1 - cnt2)

    # Обчислення відносної різниці
    if total_chars == 0:
        relative_difference = 0
    else:
        relative_difference = difference / total_chars

    # Повертаємо результати
    return  relative_difference
```

Рисунок 4.26 – Алгоритм аналізу змін

Перейдемо до аналізу швидкодії

Для них у нас буде трішки інший підхід, адже буде вираховувано та проаналізовано максимальний потенціал оптимізації таких алгоритмів. Для цього аналогічно реалізовано та додано новий метод аналізу (рис. 4.27).

```
def benchmark(self, code: str, runs: int = 10, warmup: int = 3, setup: str = 'pass'):  2 usages

    # Validate inputs
    if not isinstance(code, str):
        raise ValueError("Code must be a string")
    if runs <= 0:
        raise ValueError("Runs must be positive")

    # Warmup phase (run without timing)
    for _ in range(warmup):
        try:
            exec(setup, globals())
            exec(code, globals())
        except Exception as e:
            raise RuntimeError(f"Code execution failed: {str(e)}")

    # Measurement phase
    timings = []
    for _ in range(runs):
        start_time = time.perf_counter()
        exec(code, globals())
        end_time = time.perf_counter()
        timings.append(end_time - start_time)

    return statistics.median(timings)
```

Рисунок 4.27 – Метод аналізу швидкодії

Принцип дії цього алгоритму простий вин вимірює середній час з визначеної кількості виконання коду.

Почнемо з замірів (табл. 4.2) алгоритму пришвидшення повільного коду .

n	100000	500000	800000	1000000	1200000	1400000
Різниця	~ -406.52%	~ -16.7%	~ +31.56%	~ +45.10%	~ +49.43%	~ +55.12%

Таблиця 4.2 – Алгоритм покращення повільного коду

Всі ці вимірювання проводились на спеціально заготовленому шматку коду (рис. 4.28). Відповідно можна побачити, що при нескладних функціях додавання спеціального обрамлення не є дуже ефективним рішенням. Навіть навпаки, при n=100000, алгоритм показує значну просадку по швидкодії ПЗ, але при збільшенні складності самого коду пропорційно збільшується і потенційна оптимізація. На таблиці максимальне значення 55.12% це пов’язано з тим, що

після досягнення значення в 800000 обчислення, система на якій все це було протестовано вже почала дуже повільно обробляти, тому за гарного «заліза» та при більшій складності функції цифри можуть бути набагато більшими. Але треба також мати на увазі, що це оптимізація шматка коду, а не всього загалом.

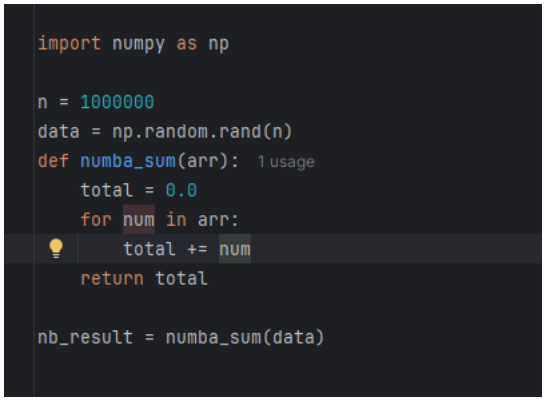


Рисунок 4.28 – Код для оптимізації

Перейдемо до вимірювання (табл. 4.3) рівня оптимізації алгоритму рекурсії

n	2	4	6	8	10	12	14	16
Різниця	–34.93%	-31.63%	-13.04%	28.21%	71.40%	91.43%	97.73%	99.42%

Таблиця 4.3 – Алгоритм оптимізації рекурсії

Також спеціально заготовлений код (рис. 4.29) використовувався для вимірювання. Загалом ми маємо дуже демонстративну версію для порівняння з варіантом з кешуванням, адже бачимо з коду при кожному крім останнього проходу функція буде викликати сама себе два рази з меншими параметрами. Почнемо аналізувати таблицю і одразу видно, що рекурсія аналогічно до алгоритму до цього спочатку показує від’ємні значення оптимізації. Це пов’язано з тим, що код при перших вимірах використовує мало ресурсів, але підключення бібліотеки сповільнює обробку, але зі збільшенням кількості проходів пропорційно збільшується і швидкодія алгоритму і максимальне

значення наближується до 100%, хоча і ніколи не стане ними, але це дуже гарний результат. Також не треба забувати, що з мінусів кешування незважаючи на гарні результати, є збільшення необхідної пам’яті та зазвичай якщо рекурсія наявна вона є маленькою частиною коду всього проєкту.

```python
def recursive_function(n):  2 usages
    # Базова умова зупинки рекурсії
    if n <= 0:
        return

    for i in range(2):
        recursive_function(n - 1)

n = 2
for outer in range(1, n):
    recursive_function(outer)
```

Рисунок 4.29 – Код для тестування оптимізації рекурсії

Передостаннім ми проаналізуємо алгоритм розкриття, аналогічно було створено спеціальний демонстративний шматок коду (рис. 4.30)

```python
j = 5
while j > 2:
    print(j)
    j -= 1
```

Рисунок 4.30 – Демонстративний шматок коду

Відповідно до результатів (табл. 4.4) бачимо, що алгоритм ефективний тільки в межах перших 5 ітерацій. Так чому ж так? Є декілька причин, але головні з них це додаткове навантаження на інтерпретатор, передбачення гілок процесорами, паралелізм та недостатня кеш пам’ять. Хоча у цього алгоритму є і перевага він може розкривати цикли не тільки для підвищення швидкодії, а і для візуалізації самого розгорнутого коду для покращення розуміння розробником.

j	2	3	4	5	6	7
Різниця	72.31%	44.90%	27.34%	13.68%	0.90%	-9.95%

Таблиця 4.4 – Виміри алгоритму

Виміри алгоритму (табл. 4.5) видалення мертвих шматків коду проводилися дуже просто, беремо 10 однакових функцій і додаємо виклики в залежності від того яку кількість коду ми хочемо, щоб видалив алгоритм.

Кількість мертвих функцій у відсотках	100%	80%	60%	40%	20%	0%
Різниця	95.60%	67.00%	45.49%	26.98%	12.05%	-0.10%

Таблиця 4.5 – Виміри алгоритму

Відповідно до результатів бачимо, як при збільшенні кількості мертвого коду в ПЗ збільшується і ефективність оптимізації. Це відбувається через природу інтерпретатора, котрому навіть без наявності виклику цього шматка коду спочатку треба його перетворити в AST та потім в байт код. Також варто зазначити, що в реальних умовах все ж таки оптимізація буде меншою, оскільки в більшості випадків мертвий код дуже рідкісна річ.

4.3 Тестування

Для тестування кожного алгоритму було зроблено по 10 «Unit» - тестів, щоб не описувати всі окремо наглядно розкажу на прототипі (рис. 4.31)

Як ми бачимо було використано бібліотеку unittest та підключено спеціальний клас «тесткейс». Загалом логіка проста в нас є самий код з трансформацією за допомогою алгоритму та потенційний результат. Ми порівнюємо їх та виводимо чи рівні вони і все це робимо для десяти об'єктів.

```python
import unittest


class TestCodeTransformer(unittest.TestCase):
    def test_1(self):
        input_code = "some code"
        expected = "result code"
        self.assertEqual(transform_code(input_code), expected)

    def test_2(self):
        input_code = "some code"
        expected = "result code"
        self.assertEqual(transform_code(input_code), expected)

    def test_3(self):
        input_code = "some code"
        expected = "result code"
        self.assertEqual(transform_code(input_code), expected)

if __name__ == "__main__":
    unittest.main()
```

Рисунок 4.31– Прототип тестування

ВИСНОВКИ ПО РОЗДІЛУ 4

У даному розділі було запропоновано комплекс методів, спрямованих на вдосконалення програмного коду систем. Детальний опис кожного підходу дозволив визначити їхню ефективність у рамках поставлених завдань.

Проведені заміри алгоритмів засвідчили їхню продуктивність, а тестування підтвердило практичну придатність запропонованих рішень.

Отримані результати демонструють, що система є функціональною та може бути успішно впроваджена у реальних умовах.

Наступним кроком потенційного дослідження може бути оптимізація окремих компонентів для підвищення загальної ефективності.

ВИСНОВКИ

В результаті проведеного дослідження було запропоновано новий комплексний підхід до оптимізації коду, що поєднує автоматичне форматування, аналіз та покращення продуктивності, генерацію коментарів та адаптацію до нестандартизованих стилістичних вимог команд. Також, додатково реалізовано зворотну трансформацію синтаксису з блокової стилізації до Python-формату та навпаки.

В результаті вимірювань системи було з’ясовано, що використання запропонованих естетичних оптимізаторів разом, покращує якість коду в середньому на приблизно 51.42%. В залежності від використовуваного алгоритму та коду, оптимізація швидкодії може сягати неймовірних значень. Для методу видалення «мертвого коду» найбільше значення – це 95.60%, для розгортки циклів – 72,31%, для алгоритму з покращенням рекурсії - 99, 42%, для пришвидшення повільного коду – 55.12%.

З практичної точки зору, використання програмного засобу дозволяє скоротити витрати часу на рутинні завдання, пов’язані з покращенням коду, підвищити його якість та забезпечити легкість подальшого супроводу. Крім того, інструмент може слугувати корисною основою для розвитку подібних рішень або як навчальний приклад для осіб, які опановують принципи ефективного програмування на Python.

СПИСОК ЛІТЕРАТУРИ

1) Autopep8 – A tool that automatically formats Python code to conform to the PEP 8 style guide [Електронний ресурс]. – Режим доступу: https://github.com/hhatto/autopep8 – Дата звернення: 11.05.2025.

2) Black – The uncompromising Python code formatter [Електронний ресурс]. – Режим доступу: https://black.readthedocs.io/ – Дата звернення: 11.05.2025.

3) cProfile — The Python Profiler [Електронний ресурс]. – Режим доступу: https://docs.python.org/3/library/profile.html – Дата звернення: 11.05.2025.

4) Flake8 – The modular source code checker for Python [Електронний ресурс]. – Режим доступу: https://flake8.pycqa.org/ – Дата звернення: 11.05.2025.

5) isort – A Python utility for sorting imports [Електронний ресурс]. – Режим доступу: https://pycqa.github.io/isort/ – Дата звернення: 11.05.2025.

6) Pylint – Python code static checker [Електронний ресурс]. – Режим доступу: https://pylint.pycqa.org/ – Дата звернення: 11.05.2025.

7) PyCharm – IDE for Professional Developers [Електронний ресурс]. – Режим доступу: https://www.jetbrains.com/pycharm/ – Дата звернення: 11.05.2025.

8) Python – Official Python documentation [Електронний ресурс]. – Режим доступу: https://www.python.org/doc/ – Дата звернення: 11.05.2025.

9) PyQt5 Documentation [Електронний ресурс]. – Режим доступу: https://www.riverbankcomputing.com/static/Docs/PyQt5/ – Дата звернення: 11.05.2025.

10) YAPF – Yet Another Python Formatter by Google [Електронний ресурс]. – Режим доступу: https://github.com/google/yapf – Дата звернення: 11.05.2025.

11) Yappi – Yet Another Python Profiler [Електронний ресурс]. – Режим доступу: https://github.com/sumerc/yappi – Дата звернення: 11.05.2025.

12) Hettinger R. Python performance tips and tricks [Електронний ресурс]. – Режим доступу: https://www.youtube.com/watch?v=CE8UIbbQ2NM – Дата звернення: 11.05.2025.

13) Gorelick J., Ozsvald B. High Performance Python: Practical Performant Programming for Humans. – 2nd ed. – O'Reilly Media, 2020. – 450 p.

14) Van Rossum G., Drake F.L. The Python Language Reference Manual. – Network Theory Ltd., 2003. – 210 p.

15) Kurniawan B. Python Optimization Techniques [Електронний ресурс]. – Режим доступу: https://pythonperformance.com/optimization-techniques/ – Дата звернення: 11.05.2025.

16) Anderson D. Measuring and improving Python code performance. In: International Journal of Computer Applications, 2021, Vol. 183(45), p. 18–23. – DOI: 10.5120/ijca2021921307.

17) PEP 8 – Style Guide for Python Code [Електронний ресурс]. – Режим доступу: https://peps.python.org/pep-0008/ – Дата звернення: 11.05.2025.