Московский авиационный институт

(Национальный исследовательский университет)

**Лабораторные работы 1 - 12**

**Архитектура программной среды**

**Лаборатория 1.** Интегрированные среды разработки и их базовые отличия.

**Лаборатория 2.** Потоковая диаграмма GitHub и Архитектура среды. Создание локального (директории) и системного репозитория.

**Лабораторная 3.** Создание профайла VS Code для работы в среде Python. Виды профайлов. Их использование при распределенной разработки. Структура профайла.

**Лабораторная 4.** Разработка потоковых диаграмм, их реализация в среде VS Code: UML (Unified Model Language) – диаграмма взаимодействий.

**Лабораторная 5.** Создание ветвей GitHub для распределенной разработки и их слияние из среды VS Code.

**Лабораторная 6.** Файлы конфигураций (settings), создание User Task, настройка редактора VS Code.

**Лабораторная 7****.** Создание и работа в профиле для формирования документации.

**Лабораторная 8.** Создание и работа в профиле VS Code для Data Science. Используйте профиль по умолчанию.

**Лабораторная 9.** Создание и отладка веб-приложений с использованием расширения VS Code для браузера.

**Лабораторная 10.** Создание и работа в профиле VS Code для AI приложений: NumPy, TensorFlow и Python.

**\*Лабораторная 11.** Использование Docker для разработки и развертывания приложений.

**\*Лабораторная 12.** Использование ChatGPT для программирование прикладных задач.

|  |  |  |
| --- | --- | --- |
| Глушатов И. С. | подпись: | \_\_\_\_\_\_ |
| Семенов А. С. | подпись: | \_\_\_\_\_\_ |
| Дата: | \_\_\_\_\_\_\_ | 2024 |

**Лабораторная 1.**  Интегрированные среды разработки и их базовые отличия

IDE – интегрированная среда разработки – комплекс программных средств, которые используется для редактирования кода, дебага, тестирования, структурирования проектов, использования дополнительных плагинов для ускорения и удобства разработки.

Основными компонентами являются:

1. Текстовый редактор
2. Компилятор/Интерпретатор
3. Отладчик
4. Система управления версиями

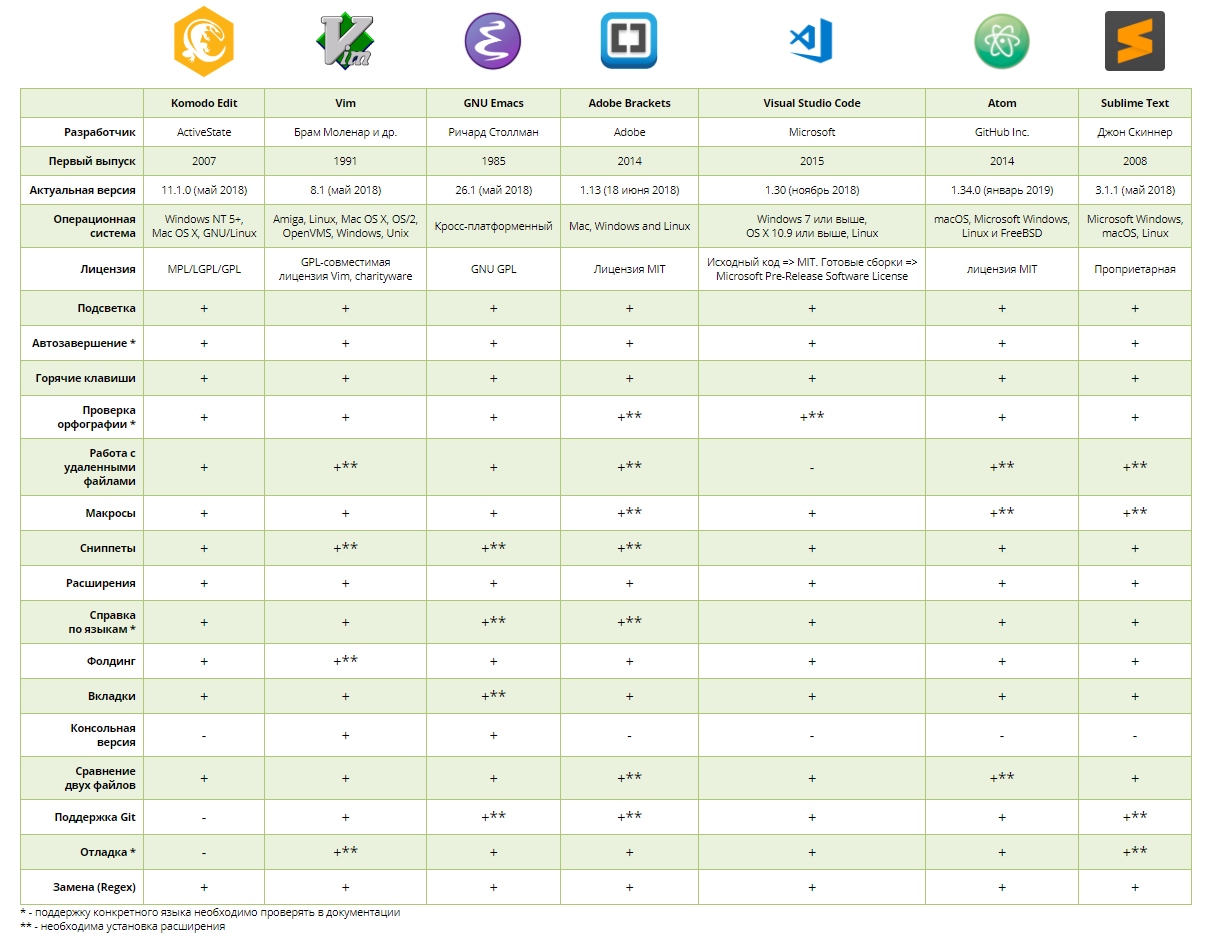
Практически все IDE поддерживают плагины.

Плагины – удобное решение, помогающее гибко подстраивать работу IDE под свои нужды. Если в редакторе отсутствует поддержка какого-либо фреймворка или языка, то это можно исправить установкой необходимых сниппетов и плагинов для поддержки синтаксиса. Если не нравится, как автоматически форматируется код и уже исчерпались возможности настроек IDE, то можно задействовать десятки различных плагинов, реализующих альтернативу встроенному форматированию.

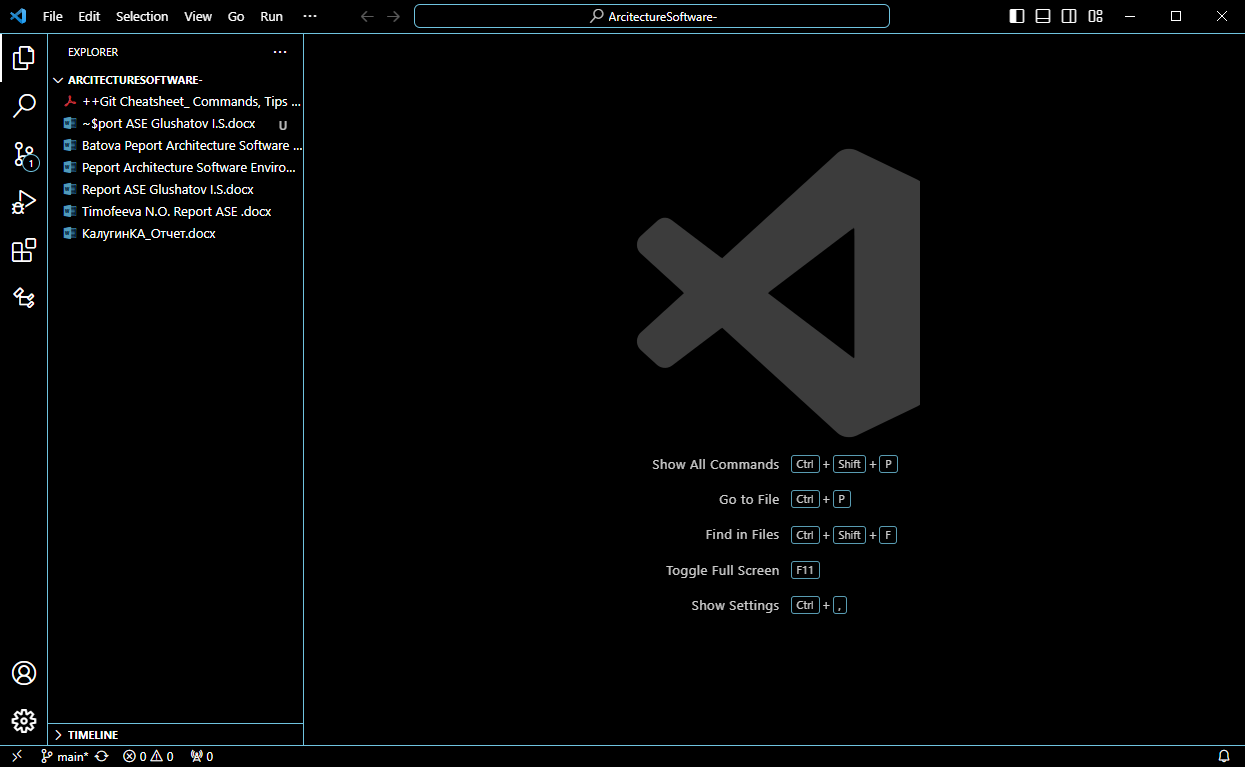
Преимущества использования IDE в разработке ПО:

1. Экономия времени – IDE помогает автоматически дописывать некоторые синтаксические конструкции; подсвечивает синтаксис, что позволяет быстрее ориентироваться в тексте кода; подсвечивает некоторые синтаксические ошибки, что позволяет не тратить время на компиляцию и многое другое.
2. Централизованность проекта – IDE позволяет собирать проект в цельный программный продукт; настраивать пайплайн разработки-сборки-развертывания проекта; система контроля версий позволяет контролировать поток разработки проекта от начала и до конца, предотвращая потерю данных и изменений в большом масштабе.

Ниже приведена таблица со сравнением возможностей различных IDE:



Visual Studio Code – среда разработки, которая построена на основе открытых источников, работает везде и имеет встроенные функции для запуска и отладки кода, а также расширения для установки новых языков, тем, отладчиков и подключения к дополнительным сервисам.



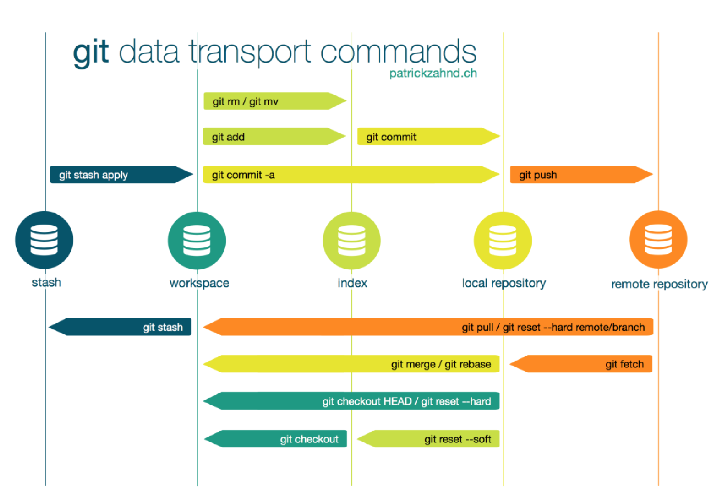
Данная IDE очень гибкая и позволяет полностью настроить под себя и свои задачи. Это можно делать путём установки расширений.

Расширения осуществляют подсветку кода, инструментами для взаимодействия с другими программами и сервисами через интернет.

Также в данной среде предусмотрен функционал быстрого поиска и настройки проекта с помощью строки.

**Лабораторная 2.** Потоковая диаграмма GitHub и Архитектура среды. Создание локального (директории) и системного репозитория.

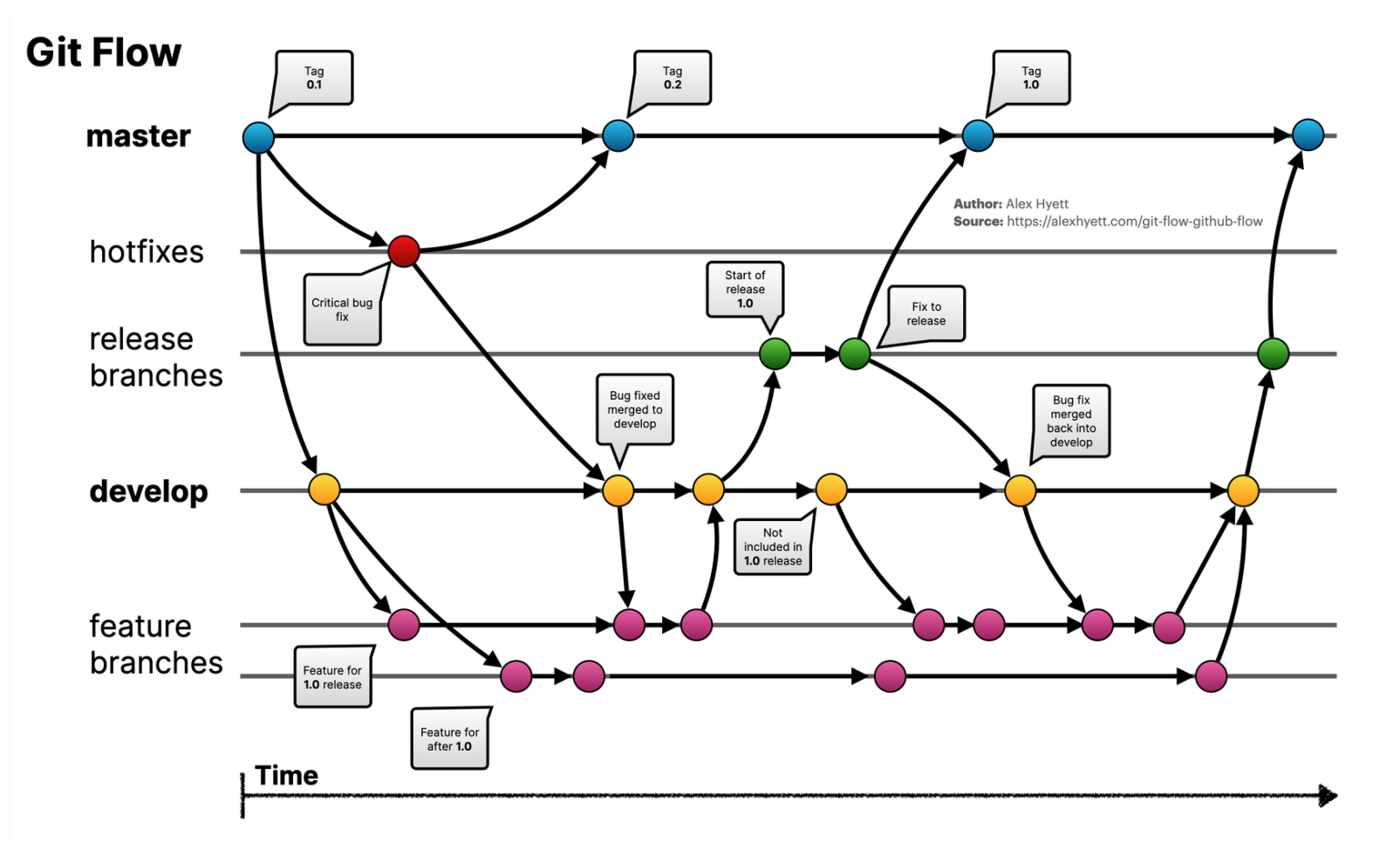
Ниже представлена общая потоковая диаграмма при работе над проектом как в локальном, так и в удаленном репозитории.



В потоковой диаграмме GitHub выделяются следующие элементы:

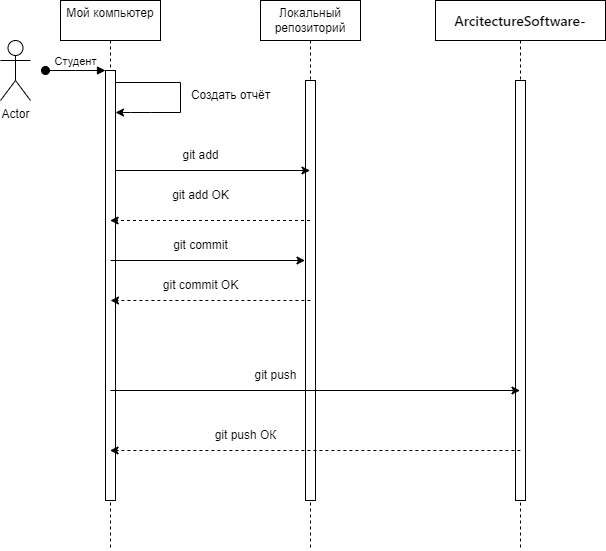
1. Workspace – рабочее пространство со всеми файлами и директориями над которыми мы работаем
2. Index – промежуточное хранилище измененных файлов перед фиксацией изменений, объединяет изменения. Не все файлы могут быть полезными(кэш, метаинформация, временные файлы, мусор), поэтому данная область позволяет зафиксировать только важные изменения.
3. Local repository – локальный репозиторий, куда коммитятся изменения, хранятся временные ветки и другая метаинформация.
4. Remote repository – централизованный удаленный репозиторий, где хранятся версии проекта, позволяет делиться изменениями между несколькими компьютерами.

Важнейшим компонентом потоковой диаграммы являются branches (ветки). Ветки нужны, чтобы несколько программистов могли вести работу над одним и тем же проектом или даже файлом одновременно, при этом не мешая друг другу. Кроме того, ветки используются для тестирования экспериментальных функций: чтобы не повредить основному проекту, создается новая ветка специально для экспериментов. В больших проектах часто используется более комплексная структура:



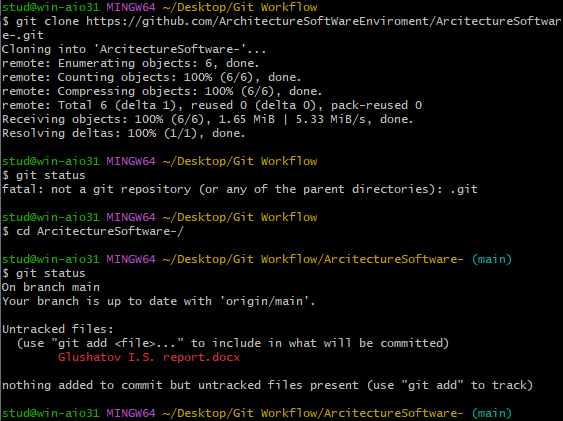
1. Master – основная ветка, которая отражает проект на стадии продакшена.
2. Develop – ветка разработки, которая указателем на HEAD отражает последние внесенные изменения, готовые к выкладке в продакшен
3. Release – поддерживает подготовку нового производственного выпуска, позволяя в последнюю минуту исправить незначительные ошибки и подготовить метаданные к выпуску.
4. Feature – ветки, в которых проводится непосредственно разработка проекта большую часть времени. Она отводится от ветки develop и сливается с ней после внесения изменений.
5. Hotfixes – ветка для исправления критических багов, попавших в продакшен. Она должна вливаться в master и develop, чтобы избежать ошибок в следующем выпуске.

Потоковая диаграмма для типичного рабочего процесса разработки ПО:

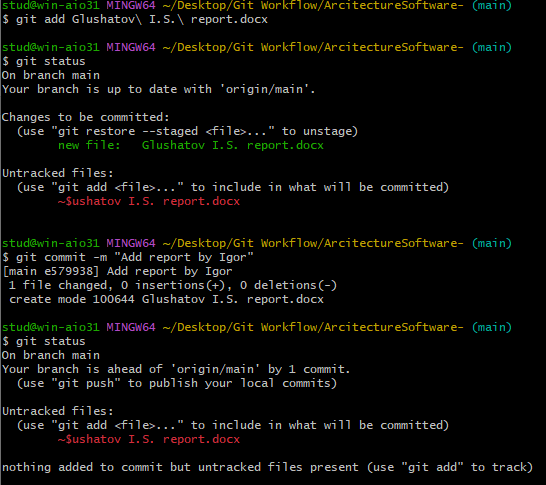


Ниже будет показан пример прохождения вышеуказанных этапов.

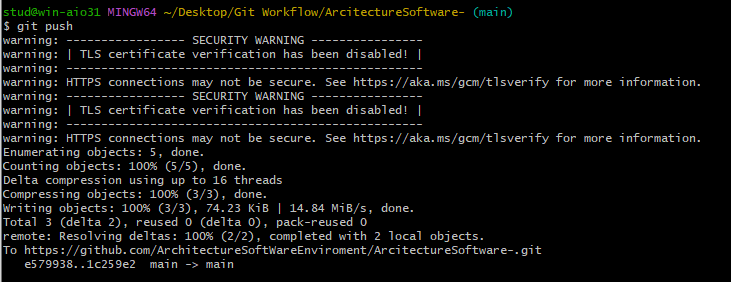
После клонирования репозитория и добавления нового файла:



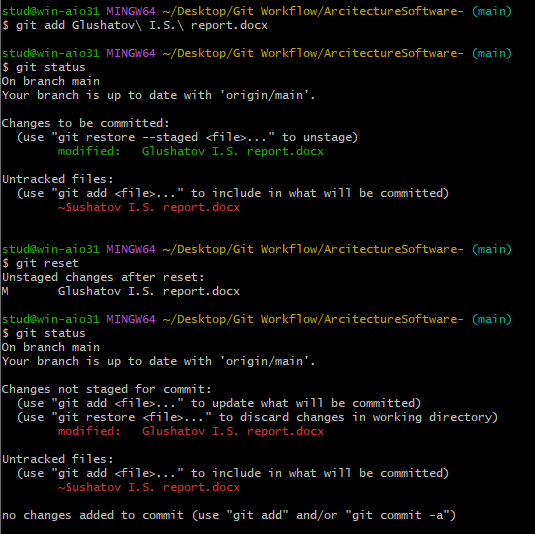
Состояние после команды add и commit:



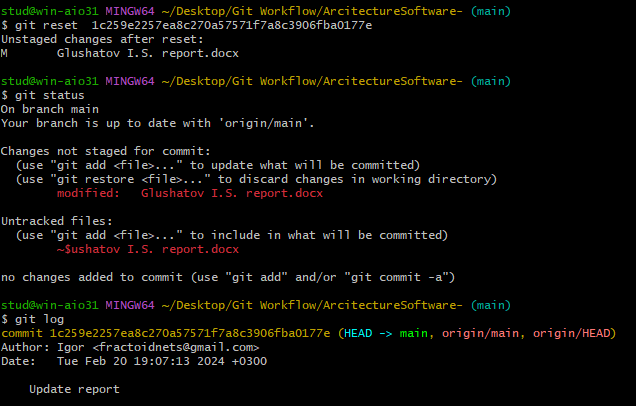
После команды push и отправления изменений на удаленный репозиторий:



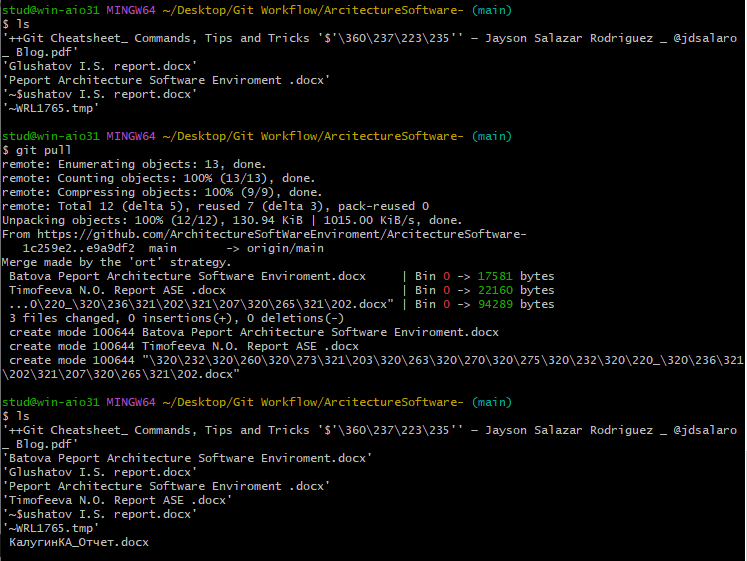
После команды add выполняем команду reset и отменяем добавление в индекс:



После команды git reset [коммит] состояние восстанавливается до указанного коммита:



После изменений на удаленном репозитории делаем команду pull:

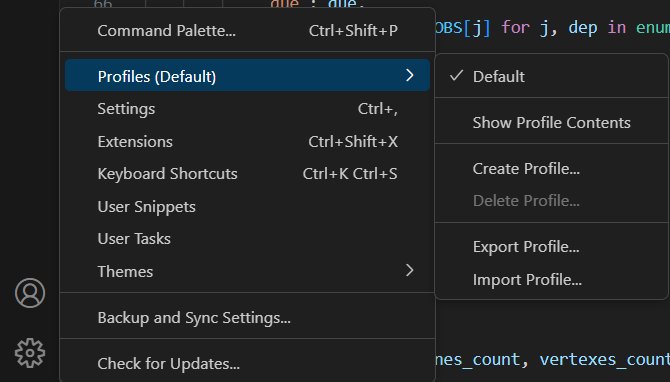


**Лабораторная 3.** Создание профайла VS Code для работы в среде Python. Виды профайлов. Их использование при распределенной разработки. Структура профайла.

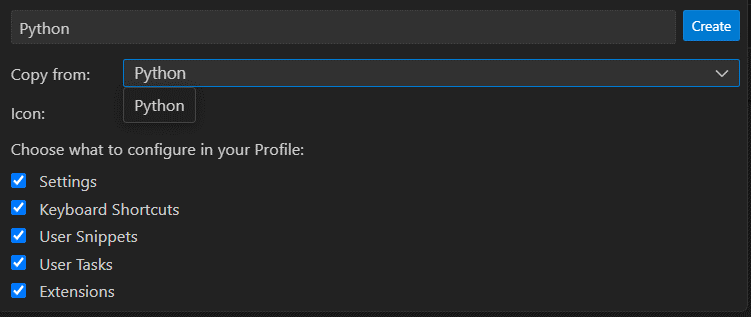
Профайл в VS Code – это набор множества настроек VS Code, плагинов и UI, который позволяет организовывать процесс разработки и делиться настройками с другими людьми.

Также бывают частичные профайлы, позволяющие менять только определенные настройки и наследовать остальные от базового профиля.

Создание профайла:

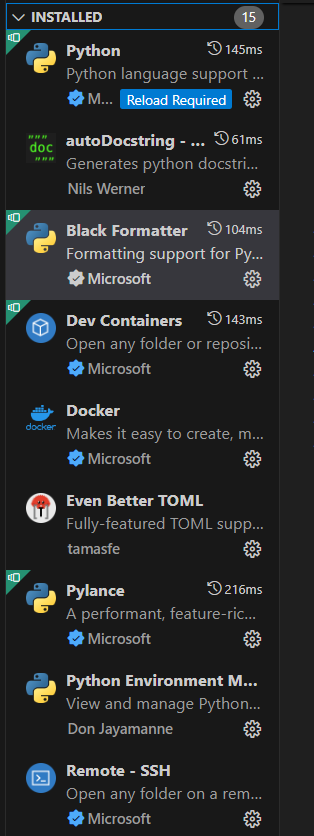
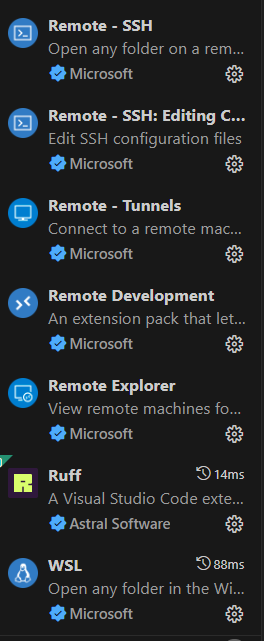


Задаем название для создаваемого профиля и выбираем корректируемые параметры. Надо ввести имя для профайла, всё остальное можно оставить по-умолчанию.

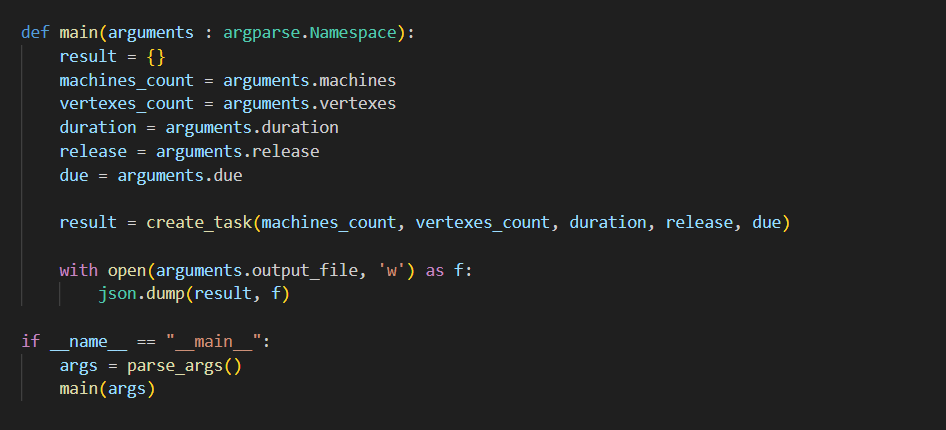


После этого видим, что в созданном профиле есть базовая часть и настройка расширений

Устанавливаем расширения для работы с Python проектом

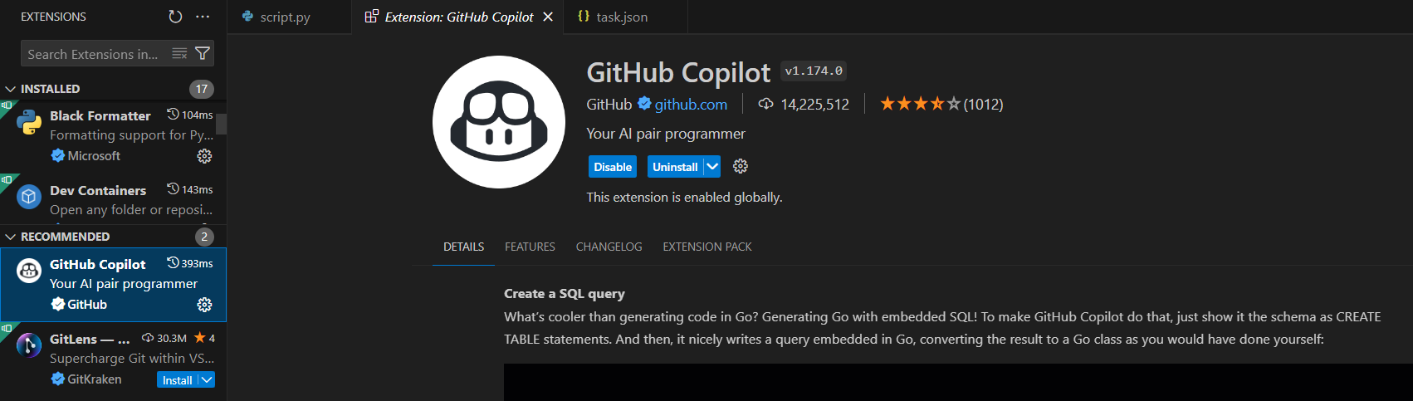
 

На написанной программе можно проверить, что всё работает.

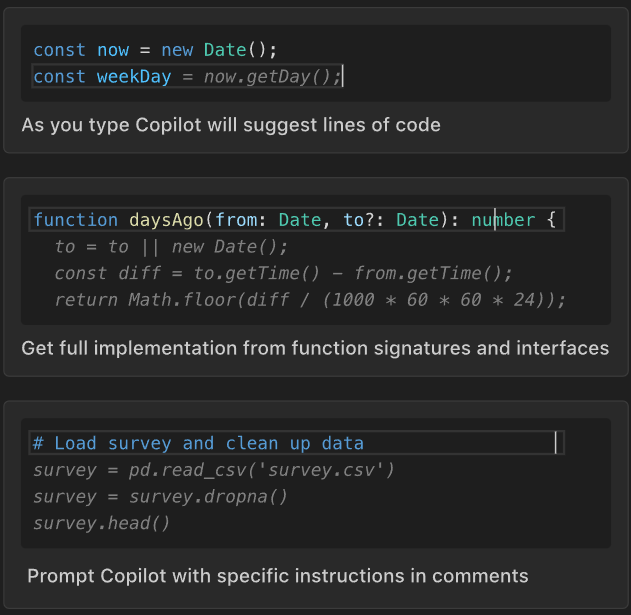


Как видим расширение работает, подсветка синтаксиса есть. Таким образом, можно сделать вывод, что создание профайлов это, по сути дела, гибкая настройка IDE для конкретного проекта или направления. Т. е. можно создать проект для веб разработки на C++, либо профиль для фронтенд разработки. При переходе на свой профиль разработчик будет чётко понимать, что у него есть всё необходимое для выполнения задачи.

Теперь установим Github Copilot.



GitHub Copilot — ваш искусственный напарник-программист. Система в реальном времени анализирует код, который пишет пользователь, а затем предлагает варианты его продолжения в виде отдельных фрагментов или целых функций. Пример работы GitHub Copilot представлен ниже.



**Лабораторная 4.** Разработка потоковых диаграмм, их реализация в среде VS Code: UML (Unified Model Language) – диаграмма взаимодействий.

Потоковая диаграмма – это диаграмма, наглядно отображающая течение информации в пределах процесса или системы. Основные её компоненты: **process, data flow, warehouse, terminator**.

UML – это унифицированный язык графического описания для объектного моделирования в области разработки программного обеспечения, для моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

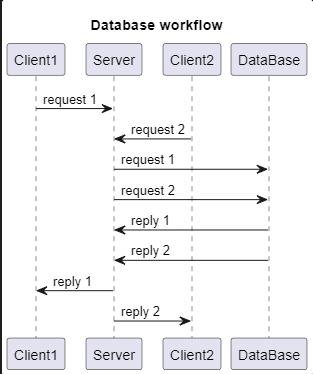
Диаграммы взаимодействия в UML представляют собой модели, которые необходимы для описания поведения взаимодействующих групп объектов. Разделяются на: коммуникации, обзора взаимодействия, последовательности, синхронизации.

Диаграммы взаимодействия используются, когда мы хотим понять поток сообщений и структурную организацию. Поток сообщений означает последовательность передачи управления от одного объекта к другому. Структурная организация означает визуальную организацию элементов в системе.

Можно использовать диаграммы взаимодействия:

1. Моделировать поток управления по временной последовательности.
2. Для моделирования потока управления структурными организациями.
3. Для перспективного проектирования.
4. Для обратного проектирования.

Теперь попробуем нарисовать UML диаграмму клиент-серверной архитектуры программы на python с базой данных с помощью плагина PlantUML в VS Code.



Код диаграммы:

@startuml

title Database workflow

Client1 -> Server : request 1

Client2 -> Server : request 2

Server -> DataBase : request 1

Server -> DataBase : request 2

DataBase -> Server : reply 1

DataBase -> Server : reply 2

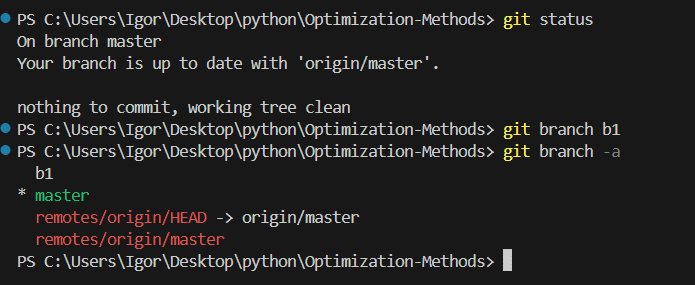
Server -> Client1 : reply 1

Server -> Client2 : reply 2

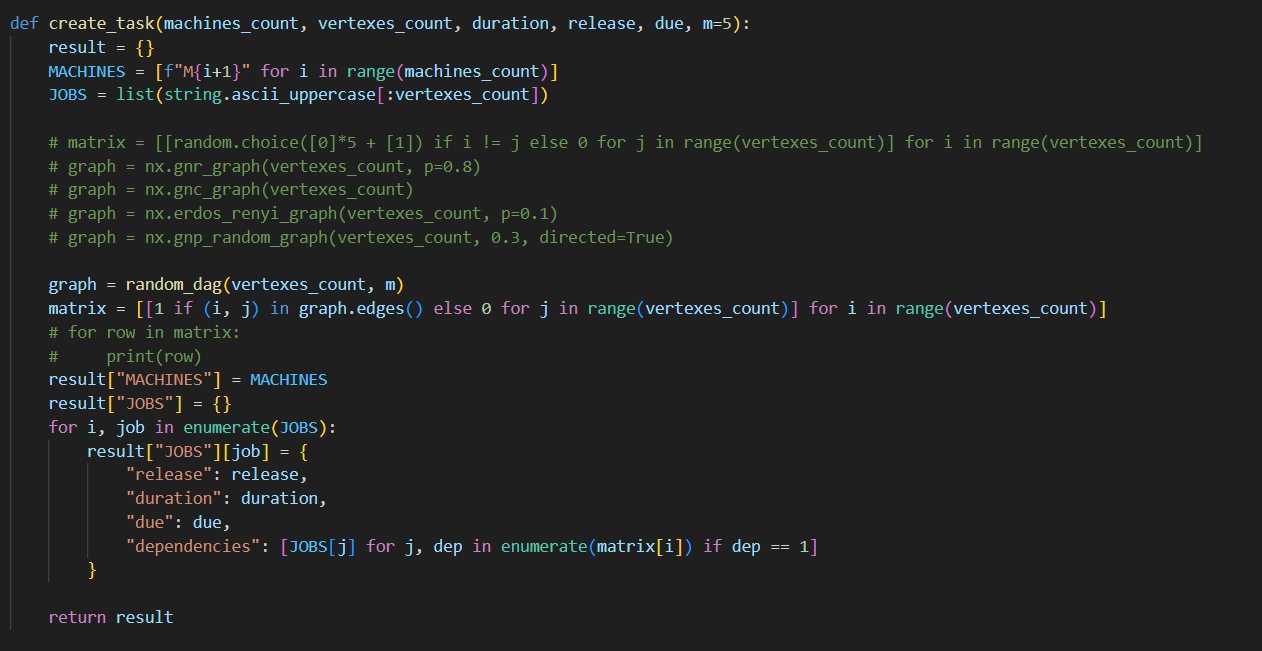
@enduml

**Лабораторная 5**. Создание ветвей в GitHub для распределенной разработки и их слияние из среды VS Code.

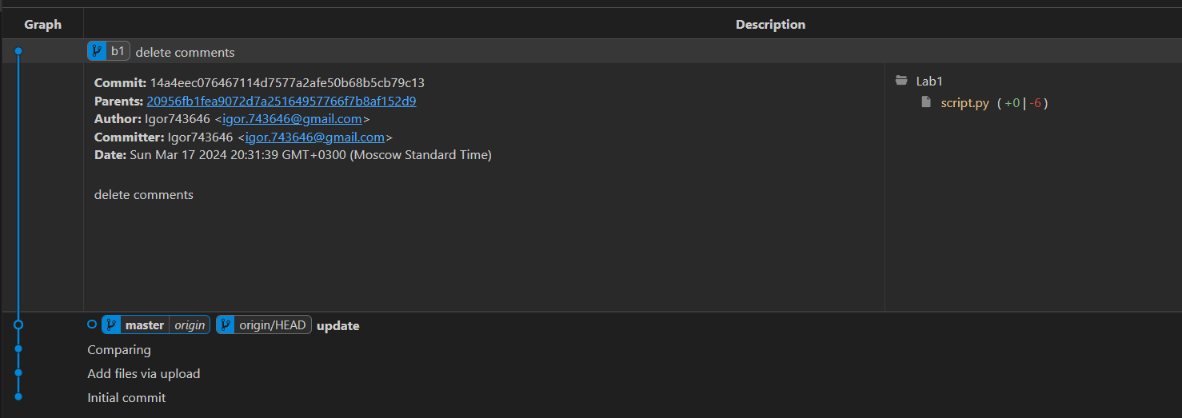
В прошлой лабораторной работе мы написали программу, которая создаем базу данный. Теперь допишем код так, чтобы была возможность добавлять данные в таблицу. Для этого создаем новую ветку и переходим в неё.



Исходный код в ветке **b1** функции **create\_task**:



В VS Code увидим изменения после перемещения в мастер ветку

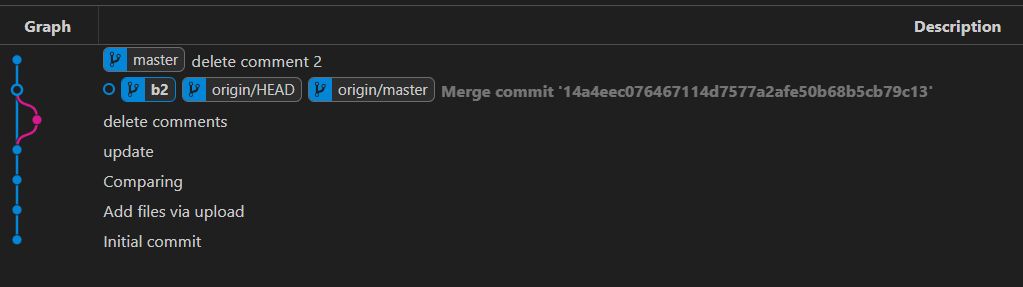


Далее можем выполнить слияние созданной ветки с мастер.

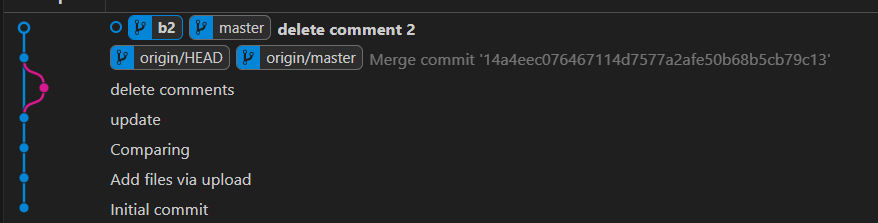
Результат слияния веток



Теперь выполним смещения статуса мастер ветки на новую ветку. Для этого создаем новую ветку и изменяем немного исходный код.



И выполняем функцию rebase branch. Результат выполнения операции rebase:

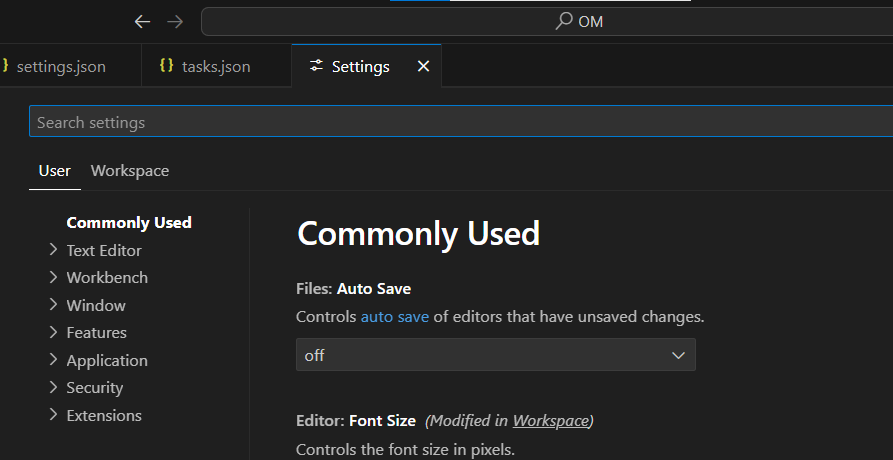


**Лабораторная 6**. Файлы конфигураций (settings), создание User Task, настройка редактора VS Code.

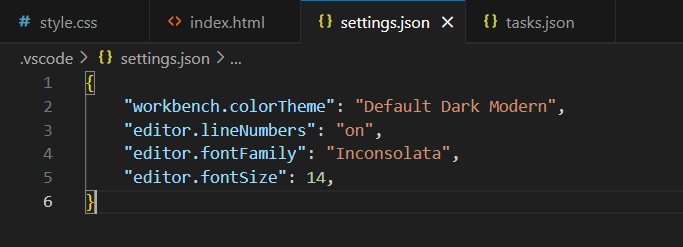
В VS Code существуют три файла конфигурации окружения:

1. Default settings – настройки по умолчанию;
2. User settings – кастомные настройки, применяемые глобально к экземпляру VS Code;
3. Workspace settings – настройки внутри рабочей области, которые применяются при её открытии.

Для редактирования пользователем разрешаются два последних. Отличаются они областью, для которой они предназначены.

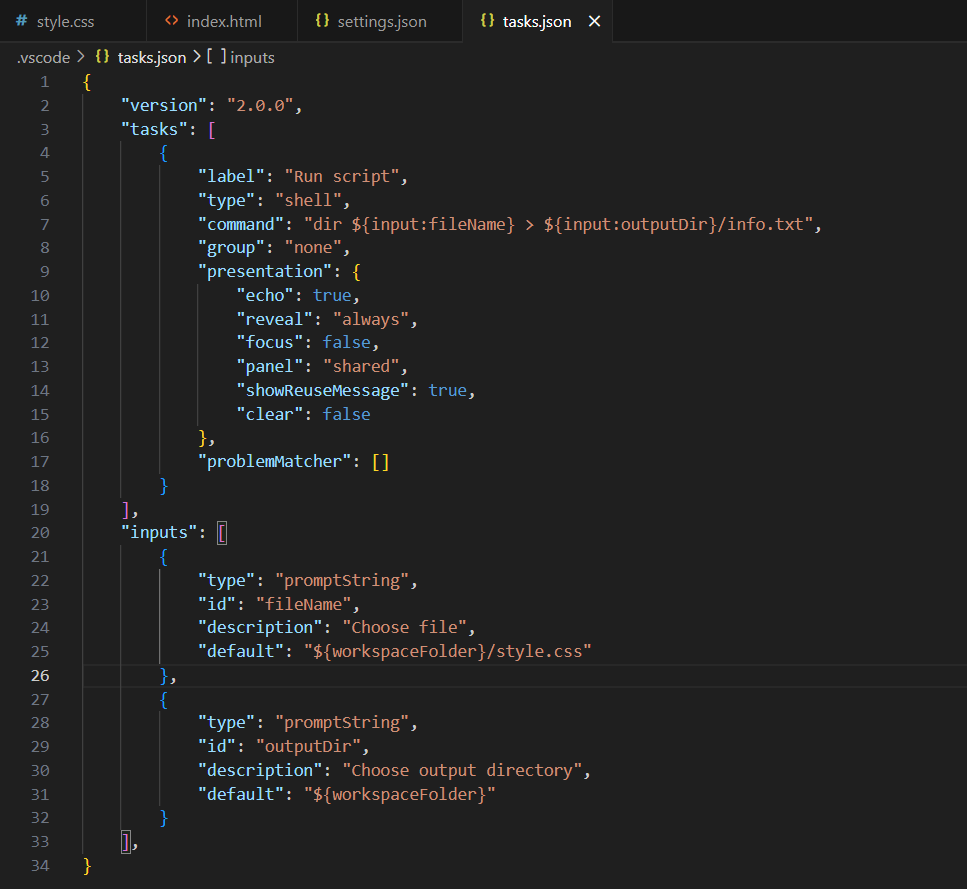


Для того, чтобы поменять настройки, в папке .vscode создадим файл settings.json и напишем туда кастомные параметры:



При сохранении изменений VSCode автоматически их применит и мы увидим изменения.

User Task – это инструмент, введенный в 2020 году, для создания пользовательских задач внутри редактора. Задачи описываются в файле .vscode/tasks.json. Можно привести пример задачи, которая собирает информацию о файле с помощью команды dir и записывает её в выходной файл. Названия файлов вводятся через Command Palette:

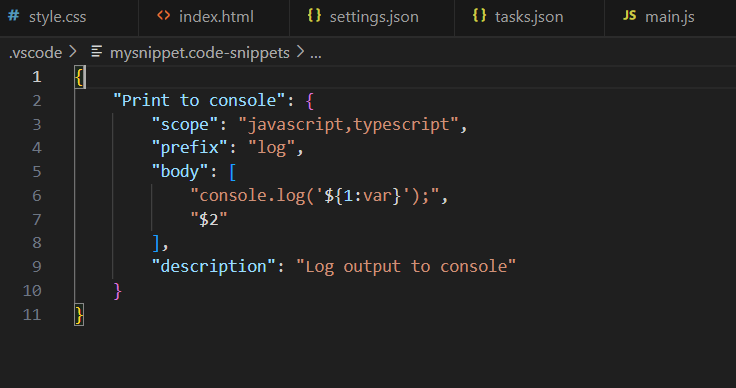


Для её запуска необходимо нажать: Выбрать панель сверху -> Run task -> Run script -> Написать путь для файла -> Написать путь на выходного файла.

Помимо вышеперечисленного в VSCode есть возможность создавать собственные сценарии для дополнения кода, называемые сниппетами. Для создания своего сниппета можно нажать на User Snippets:



Вводим имя нового сниппета и у нас откроется файл, где в формате json можно описать правило кододополнения.



**Лабораторная 7**. Создание и работа в профиле для формирования документации.

В работе с документацией полезны следующие расширения:

* Markdown All in One,
* Jupyter,
* Drawio,
* PlantUML,
* reStructuredText,
* Code Spell Checker.

Данные расширения предоставляют такие функции, как:

* подсветка синтаксиса для текстовых форматов,
* выделение опечаток и их исправление,
* генерация документации и/или компонентов документации с помощью интерфейса VS Code.

Для работы с **Markdown** можно установить расширение Markdown All in One. Оно предоставляет удобные функции для работы с Markdown, такие как автоматическая нумерация списков, встроенная проверка орфографии.

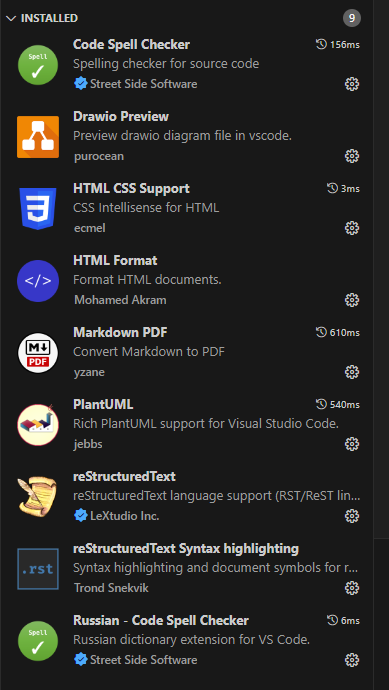
Для работы с **reStructuredText** нужно расширение reStructuredText. Оно добавляет поддержку реструктурированного текста в VS Code, включая подсветку синтаксиса и интерактивную документацию.

Для работы с **HTML** установите расширение HTML Snippets или Emmet. Эти расширения предоставляют наборы сниппетов и автодополнение кода для упрощения написания HTML.

VS Code имеет встроенную проверку орфографии, которая помогает обнаруживать и исправлять ошибки в словах. Также имеется встроенная функция автозамены, которая позволяет задать пары слов или фраз, которые будут автоматически заменяться при вводе. Это может быть полезно для сокращения часто используемых выражений или для исправления типичных опечаток.

Однако дополнительно можно установить расширение Code Spell Checker + Russian. Данное расширение позволяет находить опечатки и предлагать возможные варианты замены. Данное расширение необходимо активировать перед тем, как оно заработает.

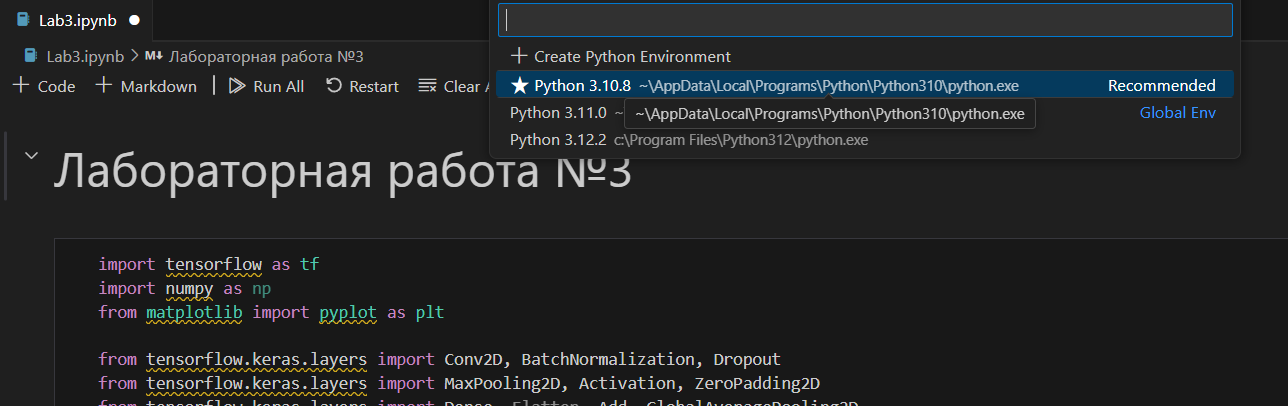
Следующий профиль получился у меня для ведения документации:



**Лабораторная работа 8.** Создание и работа в профиле VS Code для Data Science. Используйте профиль по умолчанию.

Для работы Data Science полезны следующие расширения:Python, Jupyter, GitLens, LiveShare, RemoteSSH.

Для работы с Data Science достаточно скачать данные расширения и создать файл с расширением ipynb. Для запуска кода в нём необходимо будет выбрать ядро (kernel) Python нужной версии.



Вот некоторые функции Visual Studio Code, которые могут упростить работу с данными и визуализацией для Data Science:

* **Jupyter Notebooks**. В VS Code можно редактировать блоки ноутбуков Jupyter и запускать их на сервере.
* **Data Wrangler**. Это инструмент для просмотра и очистки данных с богатым пользовательским интерфейсом. Он позволяет просматривать, очищать и анализировать данные, а также автоматически генерировать код Pandas для выполнения тех же операций.
* **Python Interactive**. В этом окне можно определять и запускать отдельные ячейки с использованием ядра IPython, визуализировать кадры данных, взаимодействовать с графиками, перезапускать ядра и экспортировать в ноутбуки Jupyter.
* **Import Jupyter Notebook**. В редакторе можно запустить команду «Import Jupyter Notebook», и код будет извлечён в файл Python.
* **IntelliSense (IntelliCode)**. Это расширение на основе искусственного интеллекта, которое предлагает варианты кода во время ввода.

Visual Studio Code (VS Code) предоставляет следующие возможности для совместной разработки проектов в области Data Science:

* **Live Share**. Этот плагин связывает VS Code с редакторами коллег, что позволяет в режиме реального времени видеть, над чем работают другие члены команды.
* **Refactoring**. Позволяет проводить рефакторинг кода.
* **Multi-file management**. Позволяет управлять несколькими файлами одновременно.
* **Git source control**. Позволяет отслеживать версии кода с помощью Git.
* **Collaborative editing**. Позволяет совместно редактировать документы.

Для управления зависимостями и виртуальными средами в Python в Visual Studio Code можно использовать следующие средства:

* **Создание виртуальных сред**. Виртуальная среда — это уникальное сочетание определённого интерпретатора Python и определённого набора библиотек. Она зависит от проекта и поддерживается в подпапке проекта.
* **Файл requirements.txt**. В виртуальной среде можно легко создать файл requirements.txt, который используется для переустановки зависимостей пакетов на других компьютерах разработки или рабочей среды. При работе в виртуальной среде для создания этого файла достаточно прописать команду pip freeze > requirements.txt.
* **Переключение между средами**. В Visual Studio можно изменить активную (текущую) среду для проекта Python в Обозревателе решений или на панели инструментов с помощью функции добавления среды.
* **Установка сторонних пакетов**. Чтобы установить сторонние пакеты, библиотеки или модули из pip или conda, необходимо активировать соответствующую среду.

Скрипт на Python для сбора и визуализации данных на Python. Пользователю надо ввести путь до csv файла, а скрипт автоматически соберет данные и распределит их на числовые и категориальные, отобразит гистограммы и корреляционную матрицу для числовых данных:

import pandas as pd

import matplotlib.pyplot as plt

# Запрос имени файла у пользователя

filename = input("Введите имя файла: ")

# Чтение данных из файла CSV

try:

data = pd.read\_csv(filename)

except FileNotFoundError:

print(f"Файл '{filename}' не найден.")

exit()

# Описание данных

print("\nОписание данных:")

print(data.describe())

# Гистограммы для числовых столбцов

numerical\_columns = data.select\_dtypes(include=[int, float])

for column in numerical\_columns:

plt.hist(data[column])

plt.xlabel(column)

plt.ylabel("Частота")

plt.title(f"Гистограмма распределения {column}")

plt.show()

# Распределение категориальных данных

categorical\_columns = data.select\_dtypes(include=[object])

for column in categorical\_columns:

plt.bar(data[column].value\_counts().index, data[column].value\_counts().values)

plt.xlabel(column)

plt.ylabel("Частота")

plt.title(f"Распределение {column}")

plt.show()

# Корреляционная матрица

if len(numerical\_columns) > 1:

correlation\_matrix = numerical\_columns.corr()

plt.matshow(correlation\_matrix)

plt.xticks(range(len(correlation\_matrix)), numerical\_columns.columns, rotation=45)

plt.yticks(range(len(correlation\_matrix)), numerical\_columns.columns)

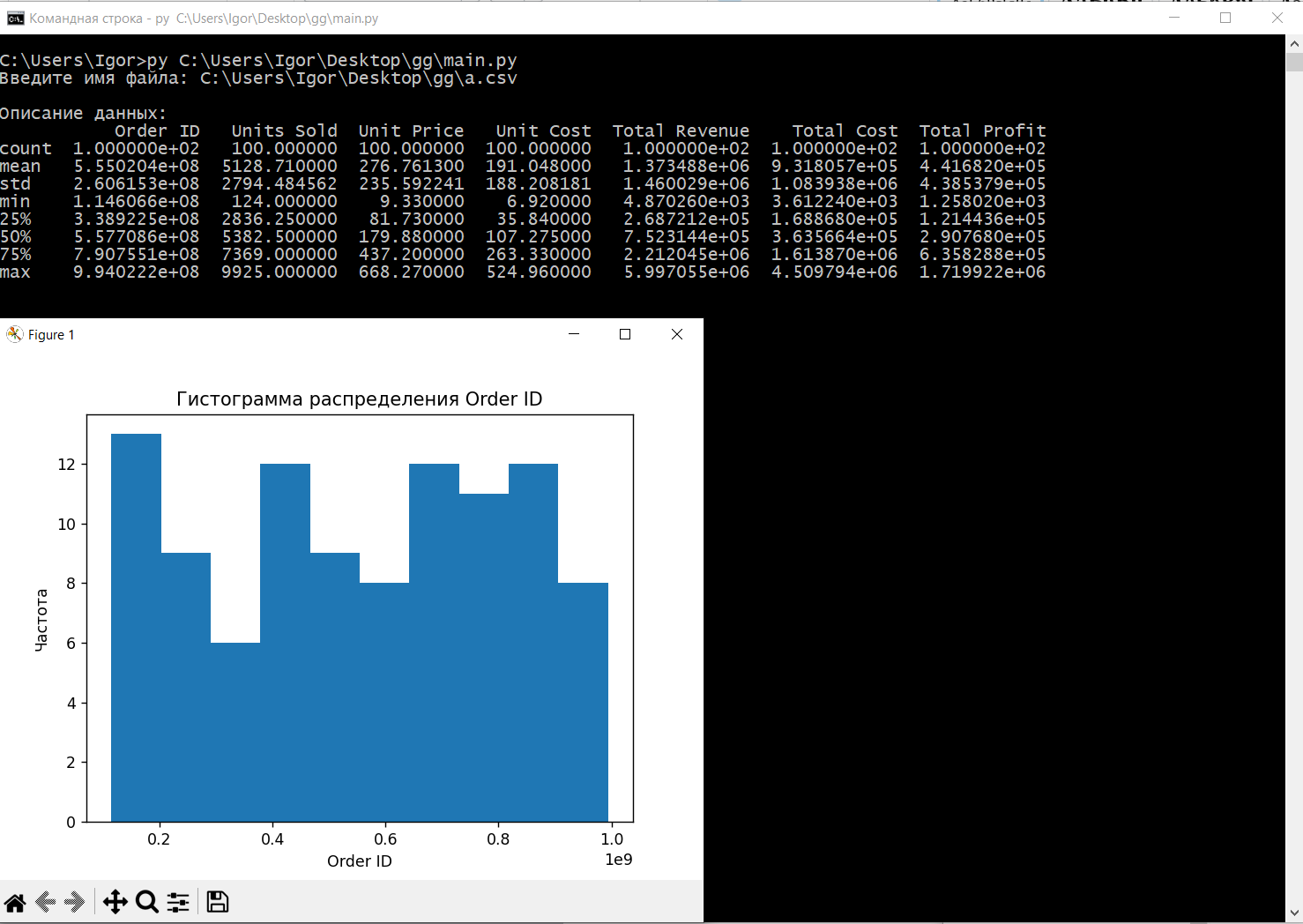
plt.colorbar()

plt.title("Корреляционная матрица")

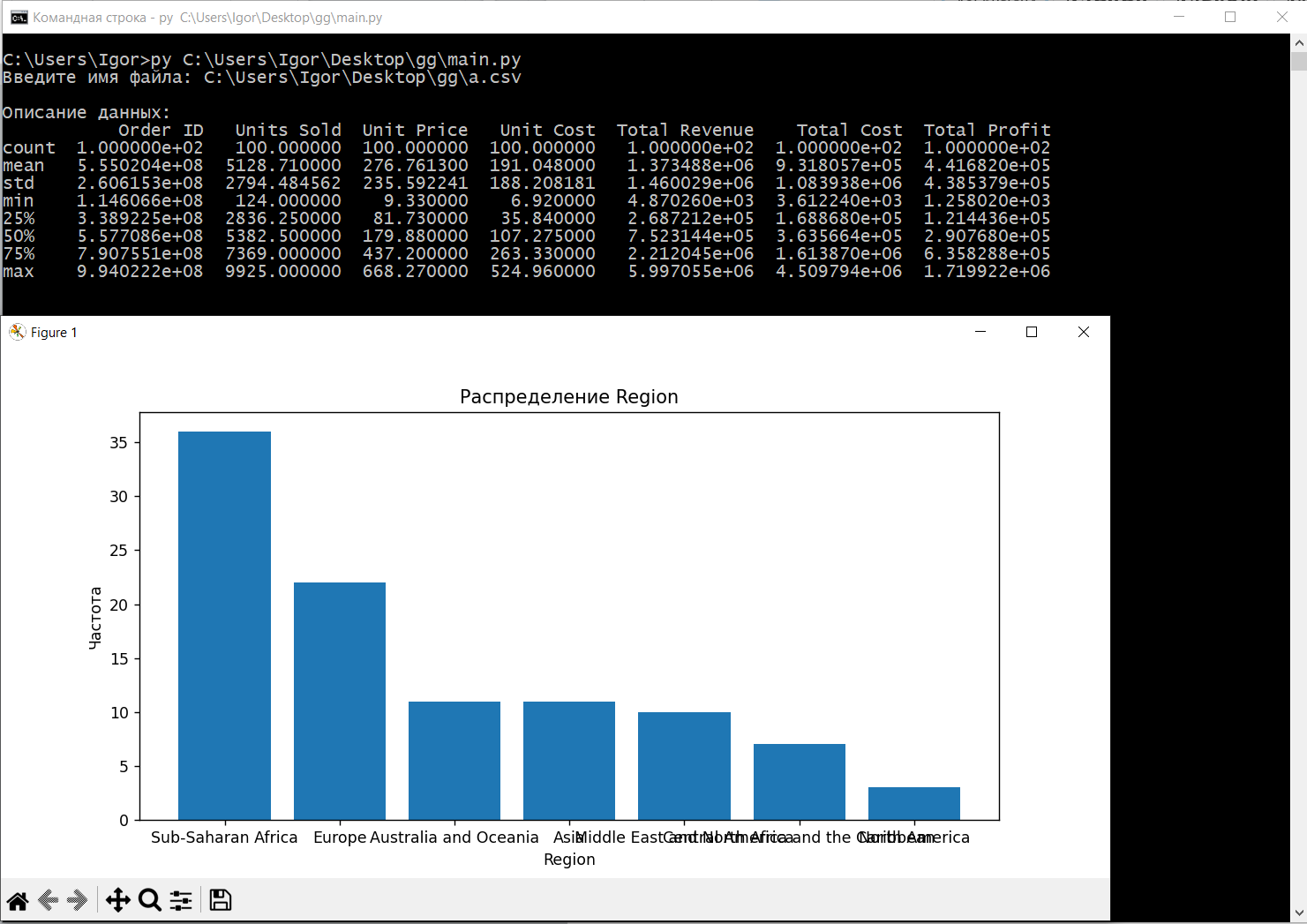
plt.show()

print("Скрипт завершен.")

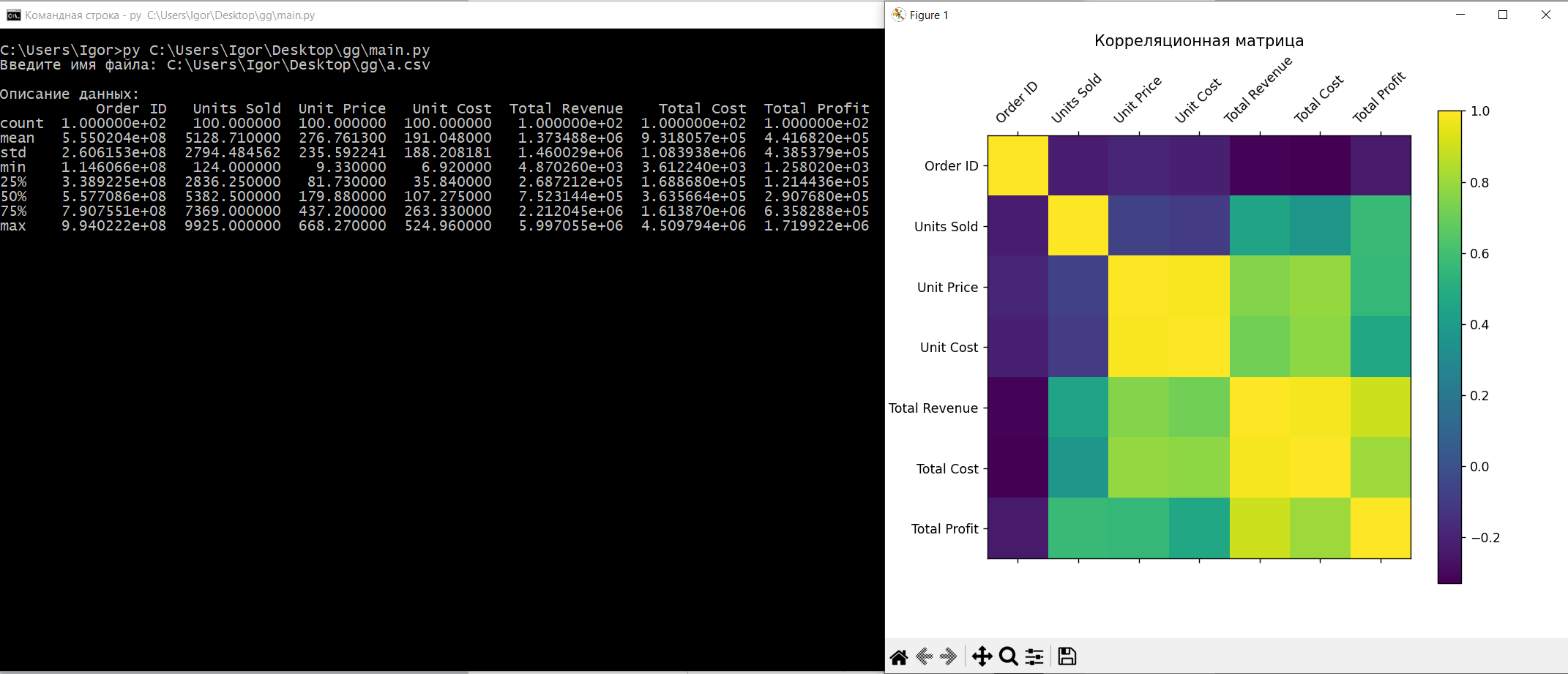
Вывод гистограммы для числового признака:



Вывод гистограммы для категориального признака:



Вывод матрицы корреляции для числовых признака:



**Лабораторная работа 9.** Создание и отладка веб-приложений с использованием расширения VS Code для браузера.

Расширения VS Code для браузера, полезные при разработке и отладке веб-приложений:

1. Расширения для отладки:

* **Debugger for Chrome**: Позволяет отлаживать JavaScript-код непосредственно в Chrome DevTools.
* **Debugger for Firefox**: Аналогично расширению для Chrome, но работает с Firefox.
* **Live Preview**: Отображает изменения кода HTML, CSS и JavaScript в браузере без перезагрузки страницы.

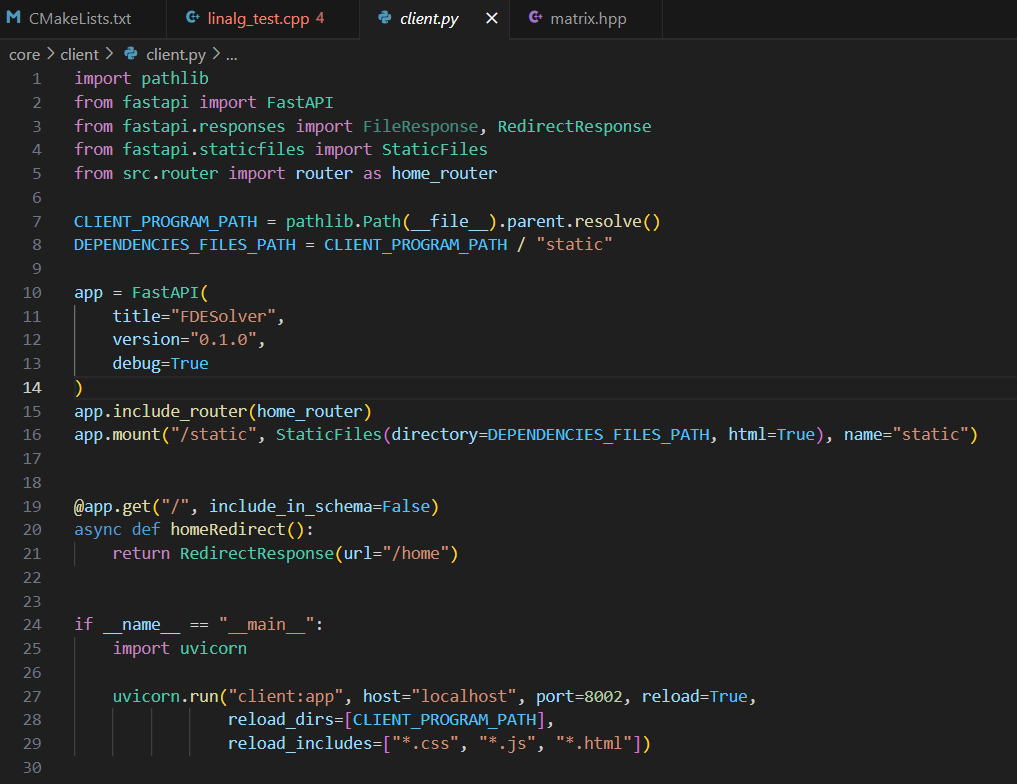
2. Расширения для работы с кодом:

* **ESLint**: Анализирует код JavaScript на наличие ошибок и несоответствий общепринятым практикам.
* **Prettier**: Форматирует код JavaScript в соответствии с выбранным стилем.
* **BEM Helper**: Помогает писать код HTML в соответствии с методологией BEM.

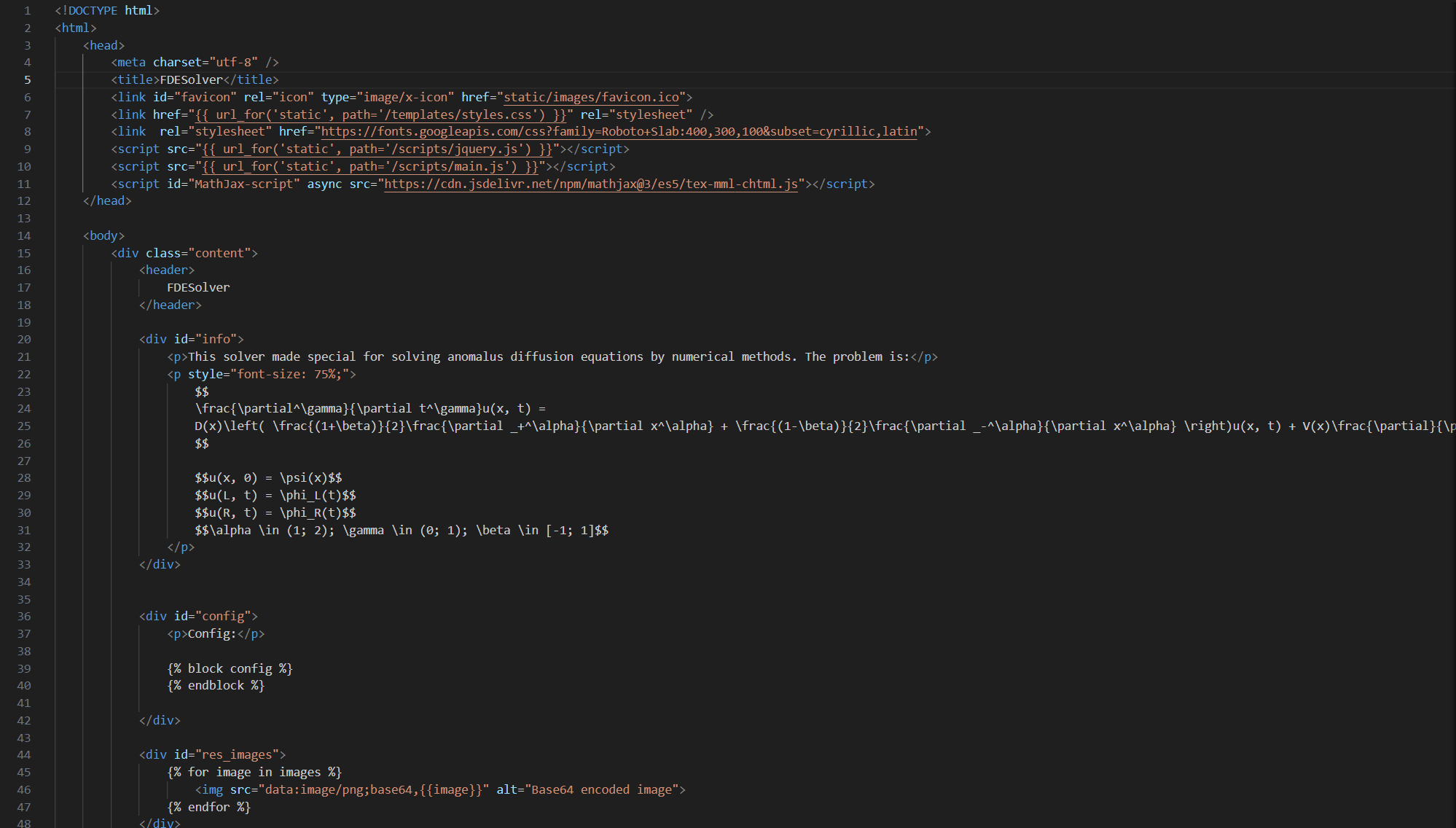
3. Расширения для повышения производительности:

* **Bracket Pair Colorizer**: Окрашивает скобки и другие парные символы в разные цвета, чтобы их было легче найти.
* **Rainbow Tags**: Окрашивает HTML-теги в разные цвета, чтобы улучшить читаемость кода.
* **CSS Peek**: Позволяет быстро просмотреть CSS-определения, наведя курсор на селектор.

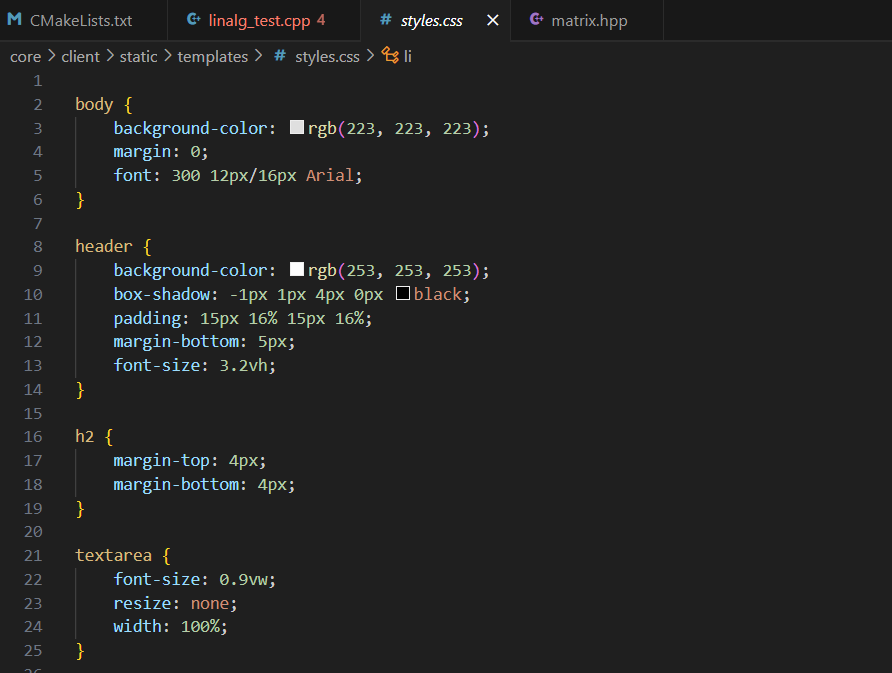
Ниже представлена часть кода на языке Python для работы веб-приложения**:**



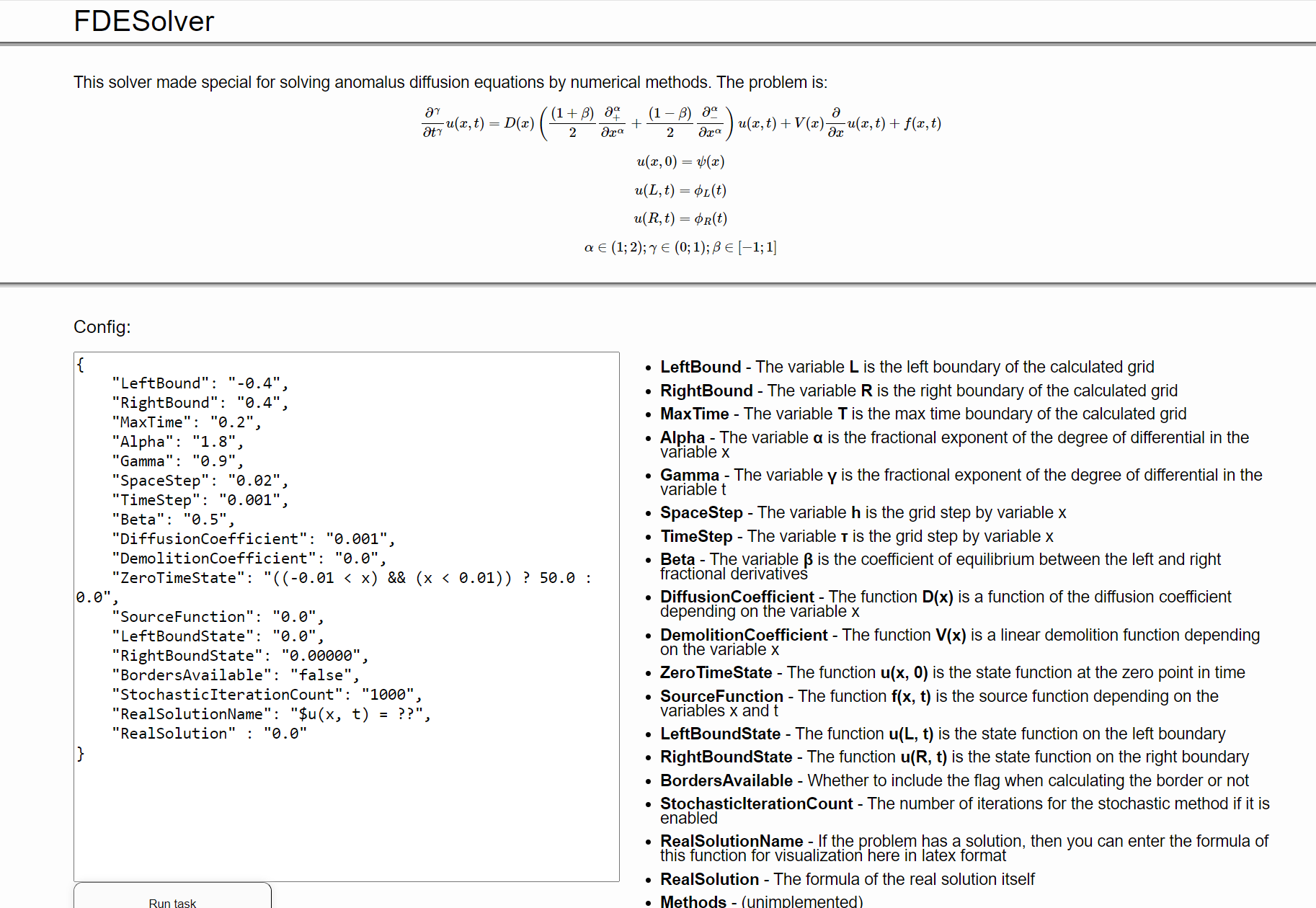
Пример HTML файла

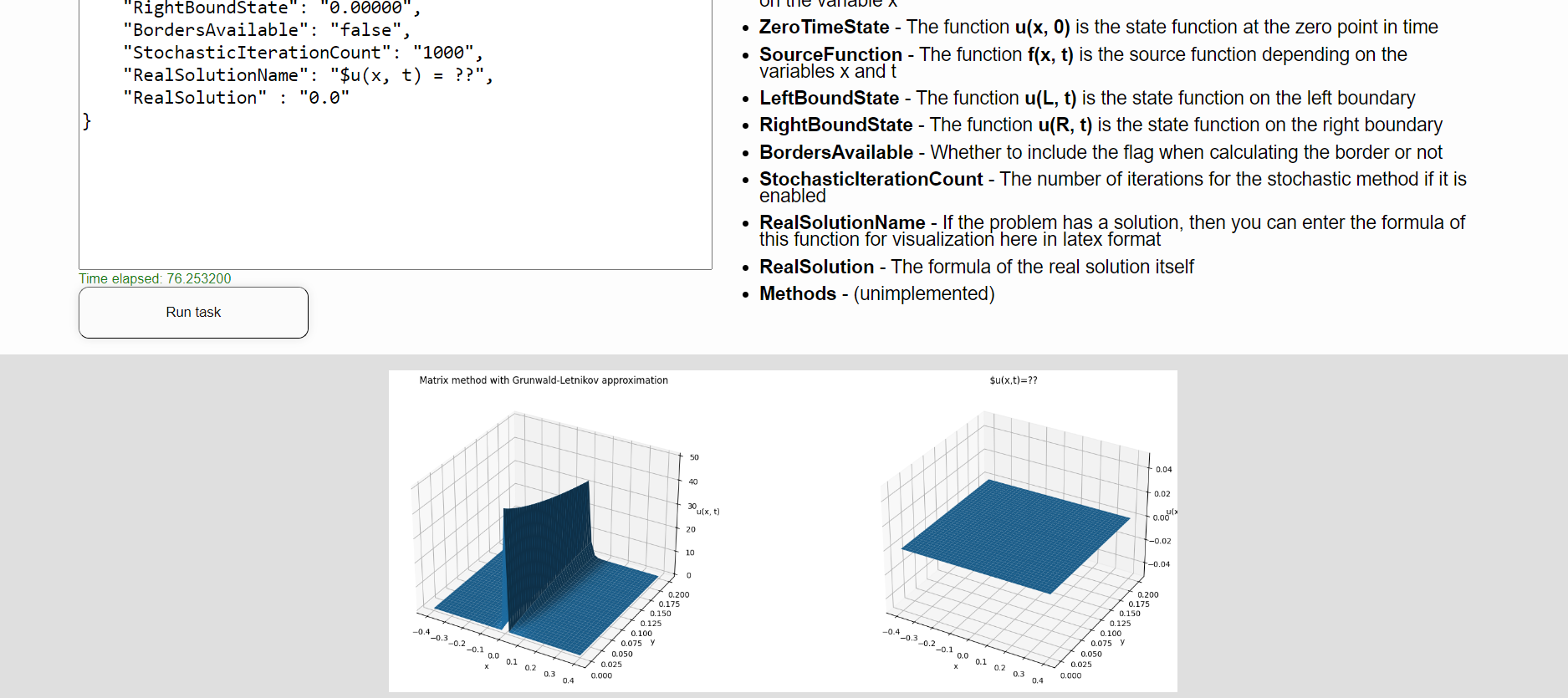


Пример CSS файла

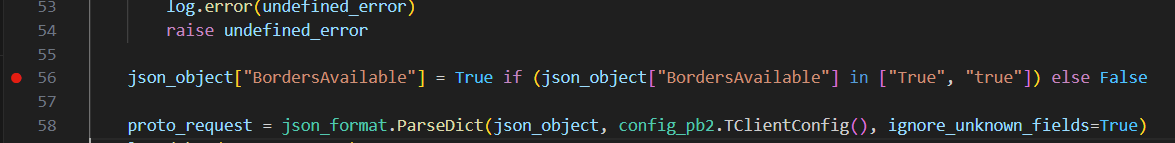


Примеры страниц

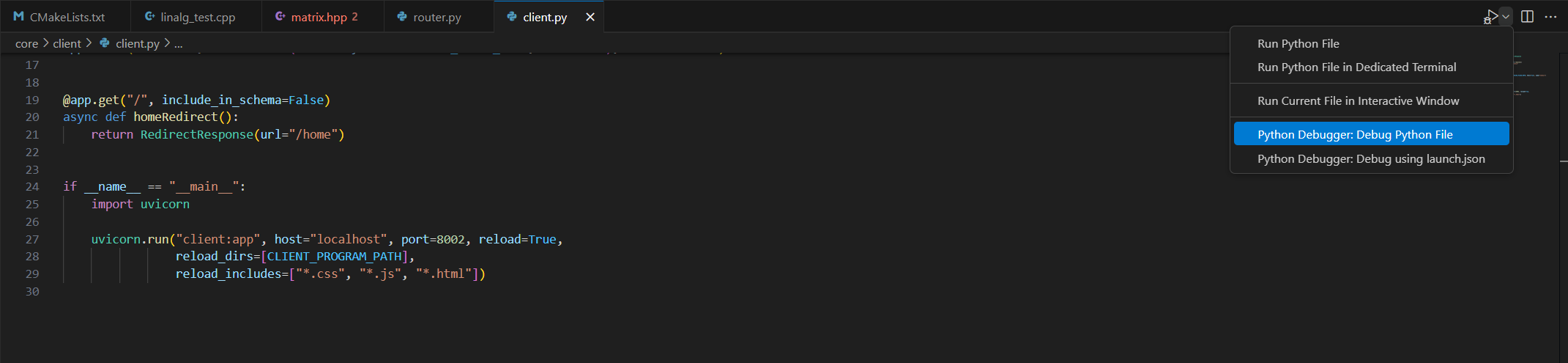




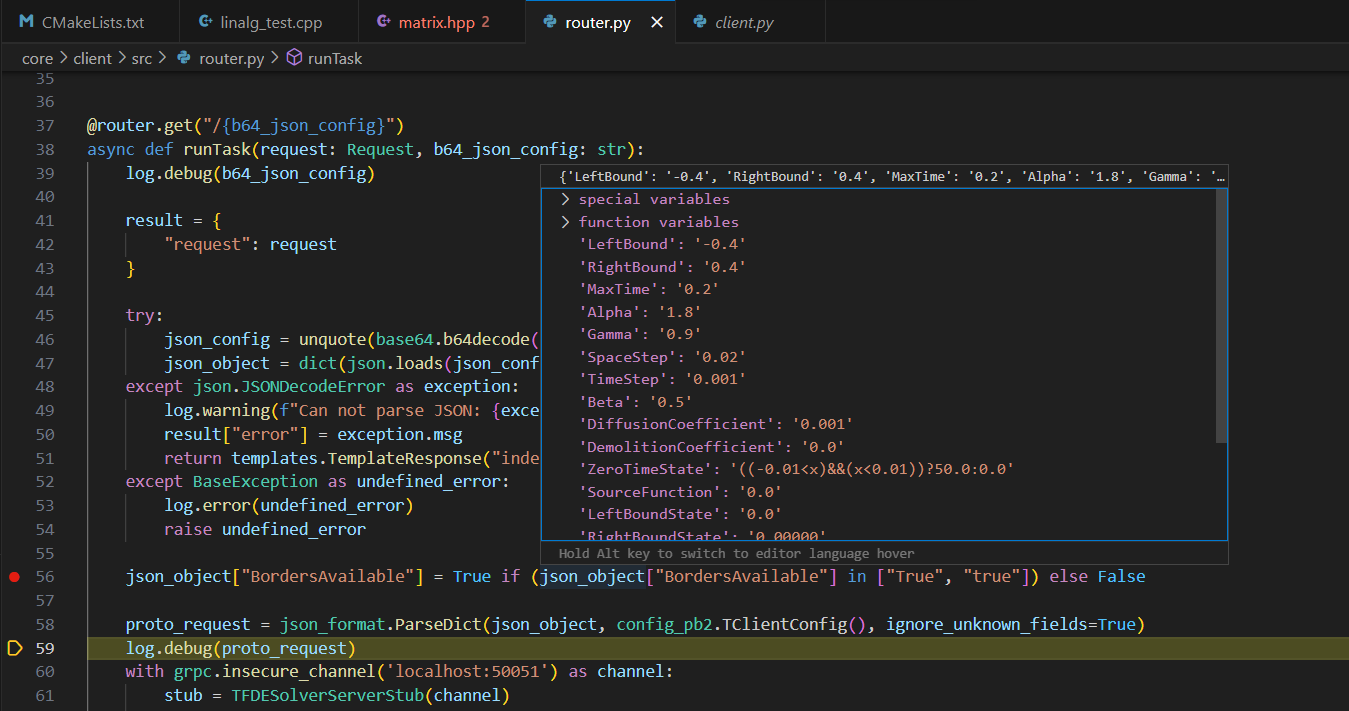
Пример работы точек остановы и проверка значений переменных.



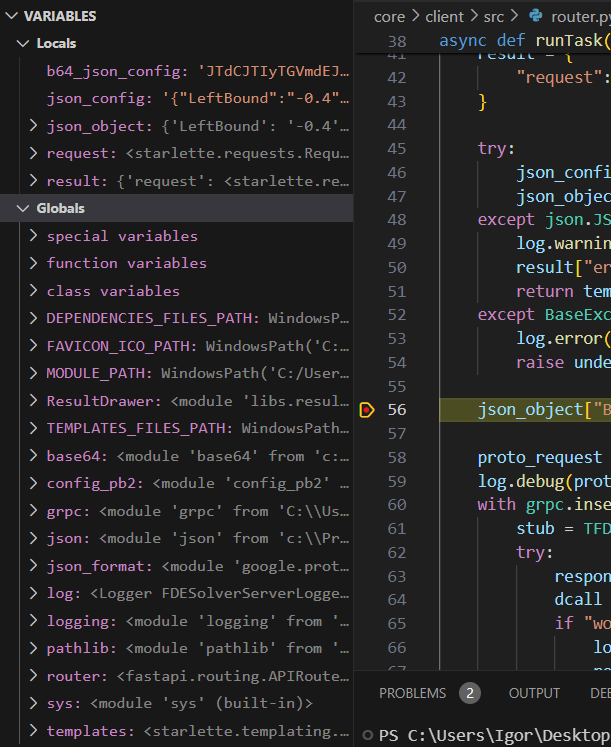
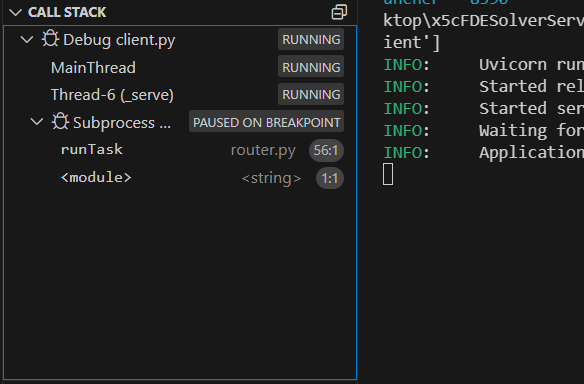
Далее запускаем отладчик



Затем вводим значения в формы и видим результат



Отображение переменных и стека вызовов:



**Лабораторная работа 10.** Создание и работа в профиле VS Code для AI приложений: NumPy, TensorFlow и Python.

Использование профиля Visual Studio Code для разработки искусственного интеллекта (ИИ) имеет следующие преимущества:

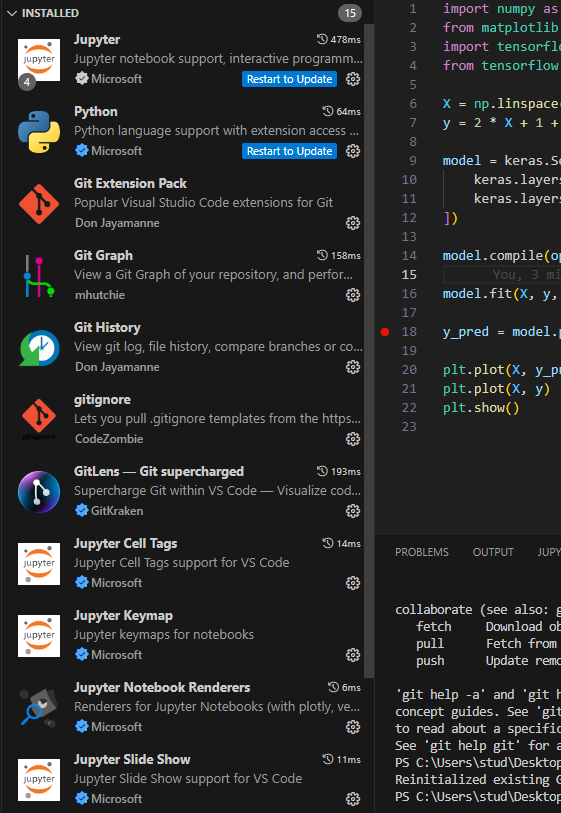
* **Создание индивидуальных сред разработки**. С помощью профилей можно настроить оптимальные настройки, цветовую тему, расширения и другие параметры для любого типа проекта.
* **Улучшение производительности**. Загружая только необходимые расширения и настройки для конкретной задачи, профили могут помочь ускорить время запуска VS Code и уменьшить использование памяти.
* **Лучшая организация**. Профили обеспечивают чёткое разделение между разными сферами работы, например, профессиональными проектами и кодом для хобби.
* **Лёгкий обмен**. Можно поделиться своей любимой настройкой VS Code с товарищами по команде или сообществом, не беспокоясь о влиянии на основную конфигурацию.

**Для** разработки искусственного интеллекта самые нужные расширения это: Python и Jupyter. Помимо этого, можно под конкретный фреймворк установить сниппеты Tensorflow Snippets, Pytorch Snippets. Никаких дополнительных настроек среды для данных расширений не надо, кроме установления версии Python интерпретатора, их достаточно скачать и пользоваться.

Вместе с расширением Python зачастую скачивается расширение Python Debugger, которое можно использовать для отладки в VS Code, ставить точки останова и смотреть данные, которые инициализированы в рабочем области видимости.

Git интегрируется аналогично остальным проектам, для этого используются расширения Git Extensions Pack, GitLens, Git Graph и другие.

Профиль, который получился у меня:



Следующий скрипт на Python обучает простую линейную модель для аппроксимации облака точек:

import numpy as np

from matplotlib import pyplot as plt

import tensorflow as tf

from tensorflow import keras

X = np.linspace(0, 10, 100)

y = 2 \* X + 1 + np.random.normal(0, 1, 100)

model = keras.Sequential([

    keras.layers.Input((1, )),

    keras.layers.Dense(1)

])

model.compile(optimizer="adam", loss='mse')

model.fit(X, y, epochs=800)

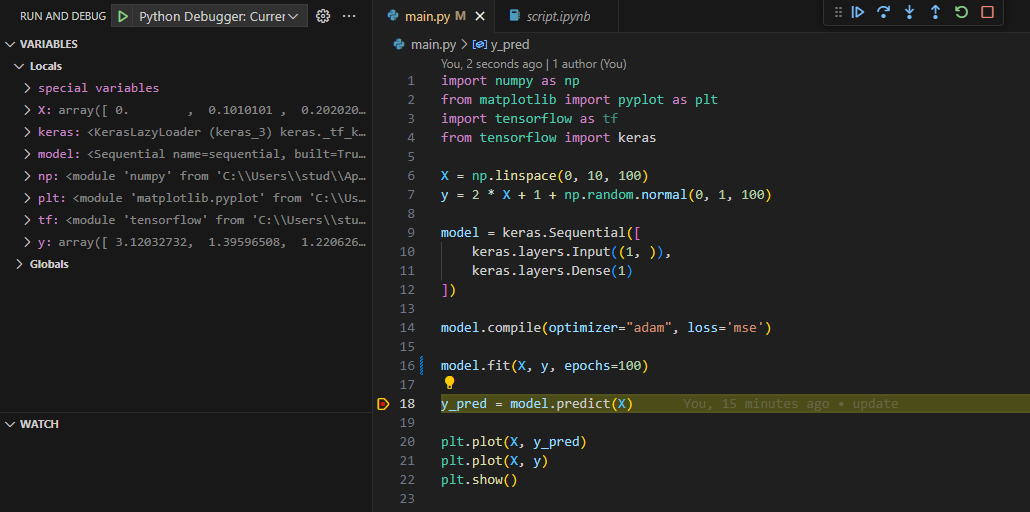
y\_pred = model.predict(X)

plt.plot(X, y\_pred)

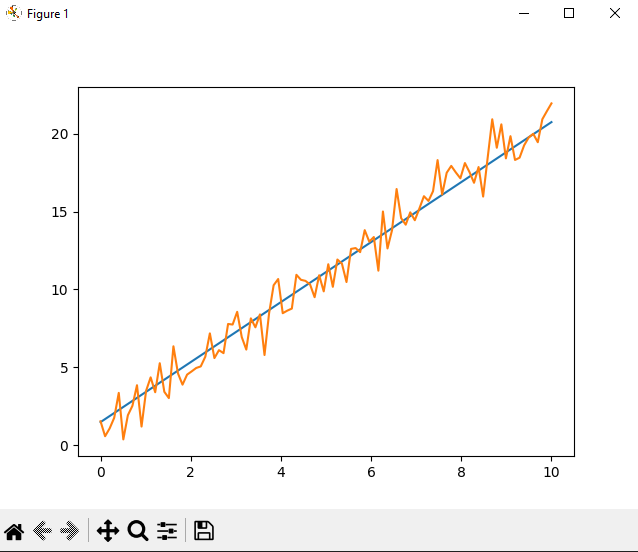
plt.plot(X, y)

plt.show()

Можно поставить точку останова и отладить скрипт:



Полученный результат:



**Лабораторная работа 11. Использование Docker для разработки и развертывания приложений**.

**Docker — это платформа и технология для контейнеризации**. Она позволяет создавать контейнеры и управлять ими для развёртывания и доставки кода на сервер.

Контейнер — это среда, внутри которой имитируется определённая операционная система. В эту систему можно положить код и запускать его в конкретной изолированной среде и в определённых условиях. Как правило, в одном контейнере запускают одно приложение или даже отдельный его компонент — модуль, функцию или микросервис.

**Преимущества использования контейнеров Docker:**

* **Возможность отката**. В любой момент контейнер можно «сбросить», откатить до изначальной версии — со всем сервером так сделать не получится. Это позволяет эффективнее тестировать приложения и легче исправлять ошибки.
* **Готовая среда для запуска**. Контейнер вместе с приложением сразу содержит среду для работы. Это позволяет загрузить готовый контейнер на любой сервер — и быть уверенным, что приложение запустится нормально.
* **Небольшой вес**. Docker-контейнер чаще всего весит не больше пары сотен мегабайтов, иногда сильно меньше. Он быстро запускается и не требует больших вычислительных мощностей.
* **Изолированность от внешней среды**. У docker-контейнера нет доступа к информации на хосте. Это делает работу приложения более безопасной.
* **Удобные инструменты управления**. Сам Docker позволяет управлять контейнерами: запускать, сохранять, редактировать, перезагружать.

Процесс развертывания приложения в Docker в простом сценарии:

1. Создание **Dockerfile**

В папке с проектом создаётся Dockerfile, являющийся инструкцией для сборки образа с описанием конфигурации и командами, которые следует выполнить для установки зависимостей.

Основные команды:

* 1. **FROM** [--platform=<platform>] <image>[:<tag>] [AS <name>] – задаёт базовый (родительский) образ. Dockerfile обязательно должен иметь инструкцию FROM!, и она должная стоять первой за исключением использования инструкции ARG.
  2. **LABEL** <key>=<value> <key>=<value> <key>=<value> ... – добавляет метаданные.
  3. **ENV** <key>=<value> ... – выставляет переменные окружения в образе при запуске.
  4. **COPY** [OPTIONS] <src> ... <dest> – копирует файлы в образ по указанному пути.
  5. **RUN** [OPTIONS] <command> ... – запускает shell команду.
  6. **CMD** ["executable","param1","param2"] – запускает основную программу, которая будет отслеживаться
  7. **WORKDIR** /path/to/workdir – задаёт рабочую директорию
  8. **EXPOSE** <port> [<port>/<protocol>...] – пробрасывает порты из виртуального окружения наружу.



После описания Dockerfile надо создать образ через команду **docker build.** В Docker Desktop можно будет увидеть строчку с созданным образом:



Для управления множеством контейнеров реализована команда **docker compose**.

Docker-compose — это надстройка над докером; приложение написанное на Python, которое позволяет запускать множество контейнеров одновременно и маршрутизировать потоки данных между ними. Конфигурирование микросервисной архитектуры осуществляется через файл **docker-compose.yml**.

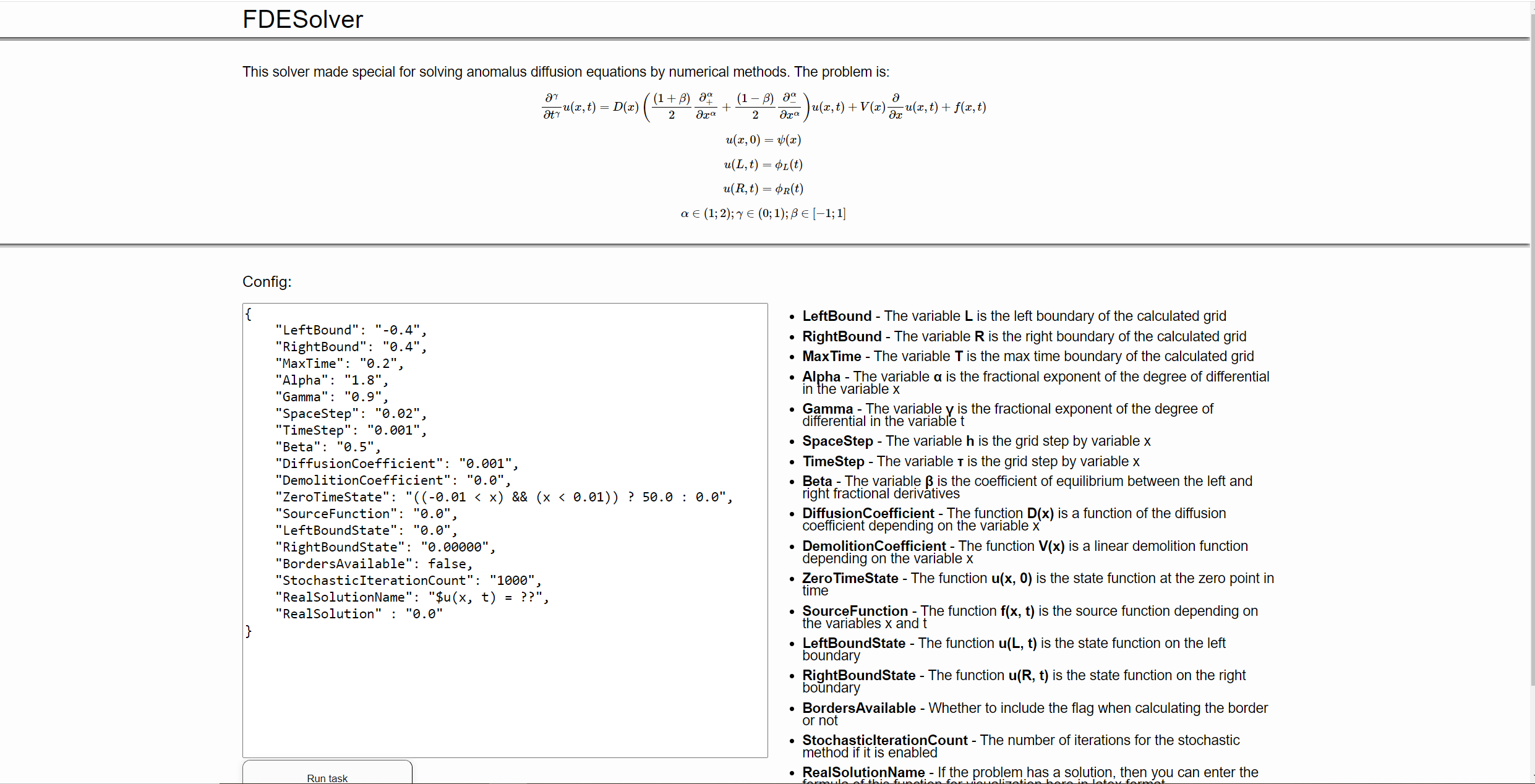
Некоторые из существующих стратегий развертывания приложений в Docker:

* Rolling – постепенная замена реплик до тех пор, пока все не заменятся.
* Recreate – простая стратегия отключения старой версии и включение новой.
* Blue-Green – запуск новой версии (Green) параллельно старой (Blue) с плавным переключением трафика.
* Canary Releases – запуск новой версии для небольшого количества пользователей для тестирования и последующей доработки и переходом на неё.
* Оркестраторы – надстройка поверх стандартных методов развертывания, выполняющая планирование, контроль ресурсов, поиск образов, конфигурирование сети, масштабирование и контролем жизненного цикла.

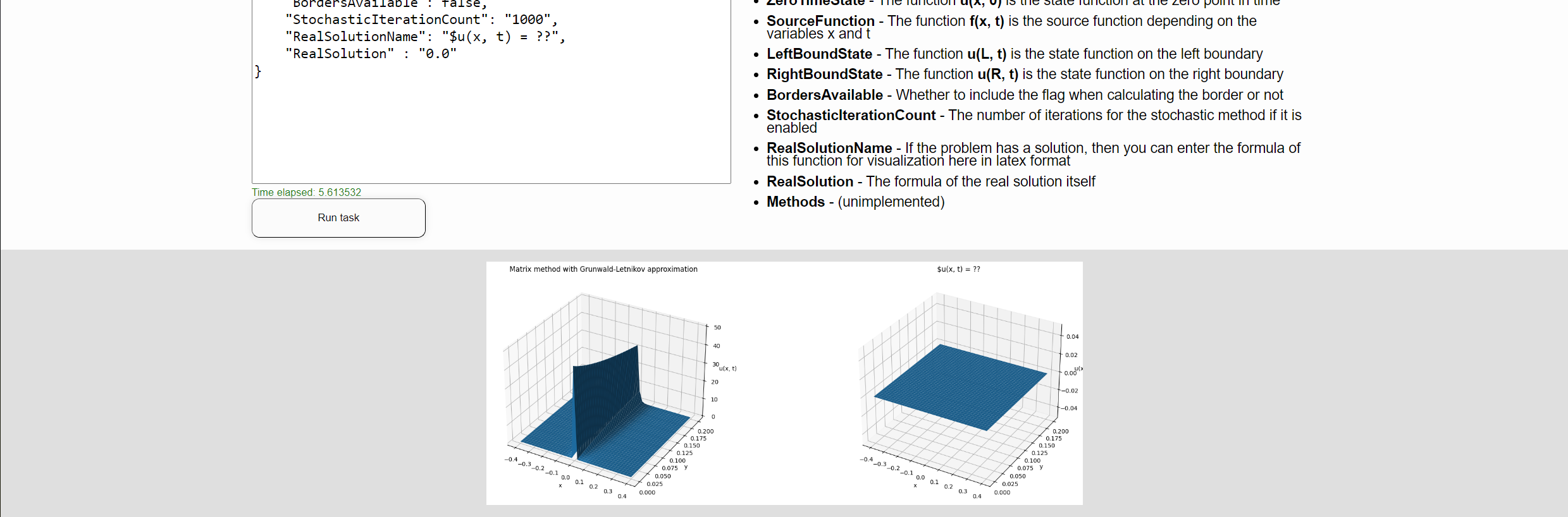
Для обеспечения безопасности и управления Docker контейнерами следует:

* Использовать официальные или собственные базовые образы при создании контейнеров.
* Внимательно анализировать образы на уязвимости посредством встроенных инструментов Docker, следить за обновлениями базовых образов.
* Не использовать тег latest при сборки образа, а указывать конкретную версию во избежание ошибок при сборки в будущем.
* Использовать секретницы, переменные окружения для инкапсуляции значимых данных.
* Использовать healthcheck для проверки контейнера на работоспособность.
* При использовании сети дополнительно действуют все меры безопасности для сетей.

Для демонстрации работы с Docker представлена клиент-серверная архитектура приложения для решения дробно-дифференциальных уравнений с web-интерфейсом. Клиент, написанный на FastApi, принимает POST запрос в виде JSON с конфигом задачи, который можно вводить в поле веб-страницы:



С помощью технологии gRPC запрос передаётся на бэкенд в виде Protobuf. Бэкенд, написанный на C++, принимает запрос, парсит входные данные, которые могут быть в том числе представлены функциями, которые анализируются с помощью библиотеки ANTLR, строя синтаксическое дерево разбора. Далее бэкенд решает данное дробно-дифференциальное уравнение, а затем отправляет ответ обратно на web-сервер, который в свою очередь обрабатывает сырые данные, преобразуя из в изображения с помощью библиотеки MatPlotLib.

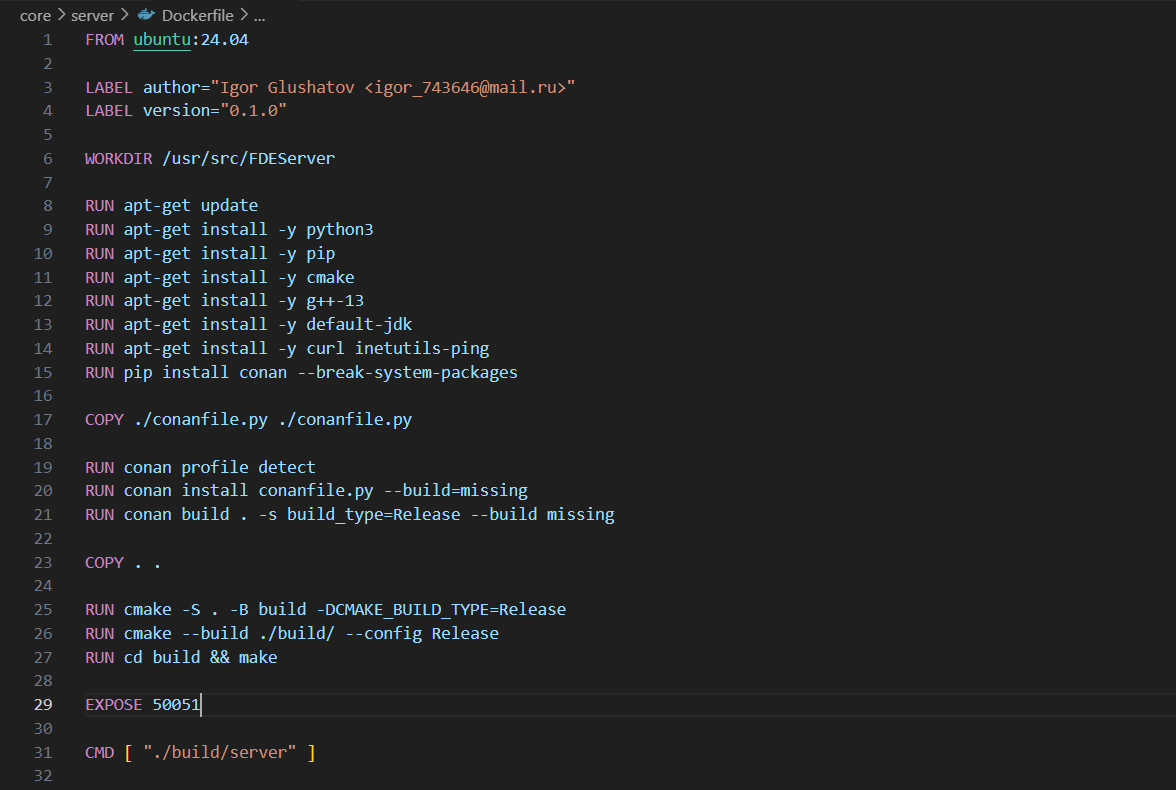


Dockerfile для развертывания простого веб-сервера на FastAPI:



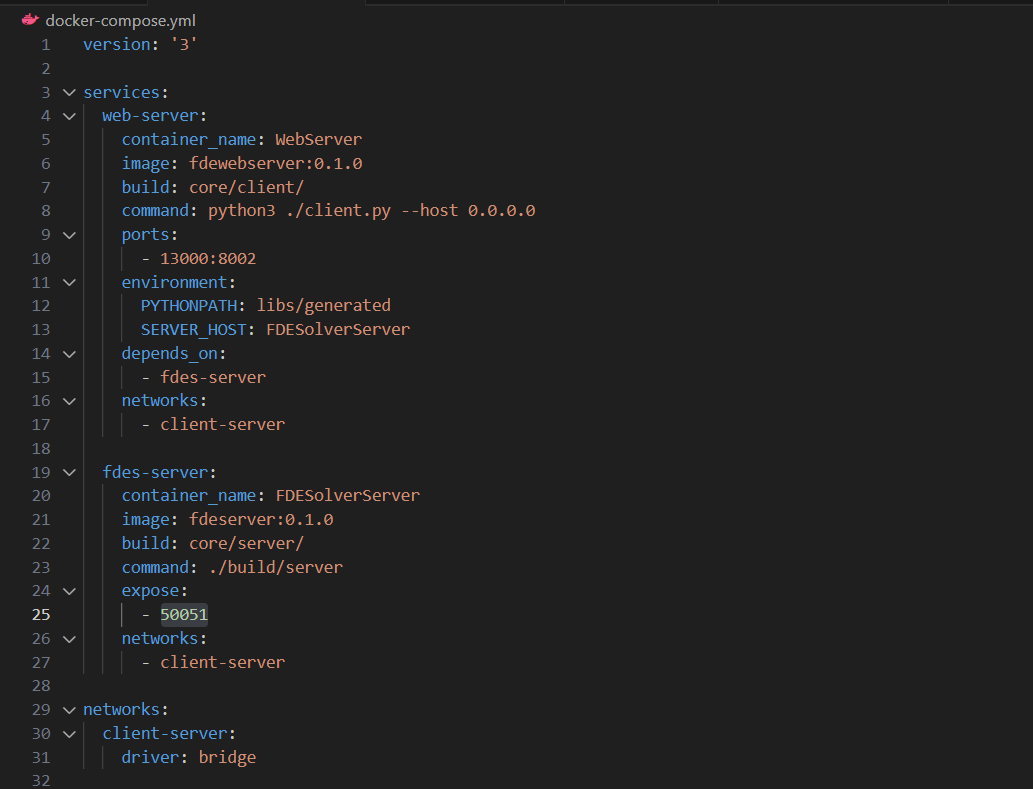
За базовый образ берется одна из последних версий Linux Ubuntu. Скачиваются пакеты для Python и утилиты для дебага работоспособности контейнера. В целях оптимизации времени сборки сначала скачиваются зависимости приложения, которые меняются редко, потом остальные файлы проекта копируются в образ. Задаётся переменная окружения для разрешения пути импорта файлов Protobuf, резервируется порт 8002 и прописывается команда запуска FastAPI приложения.

Dockerfile для развертывания бэкенда на C++:

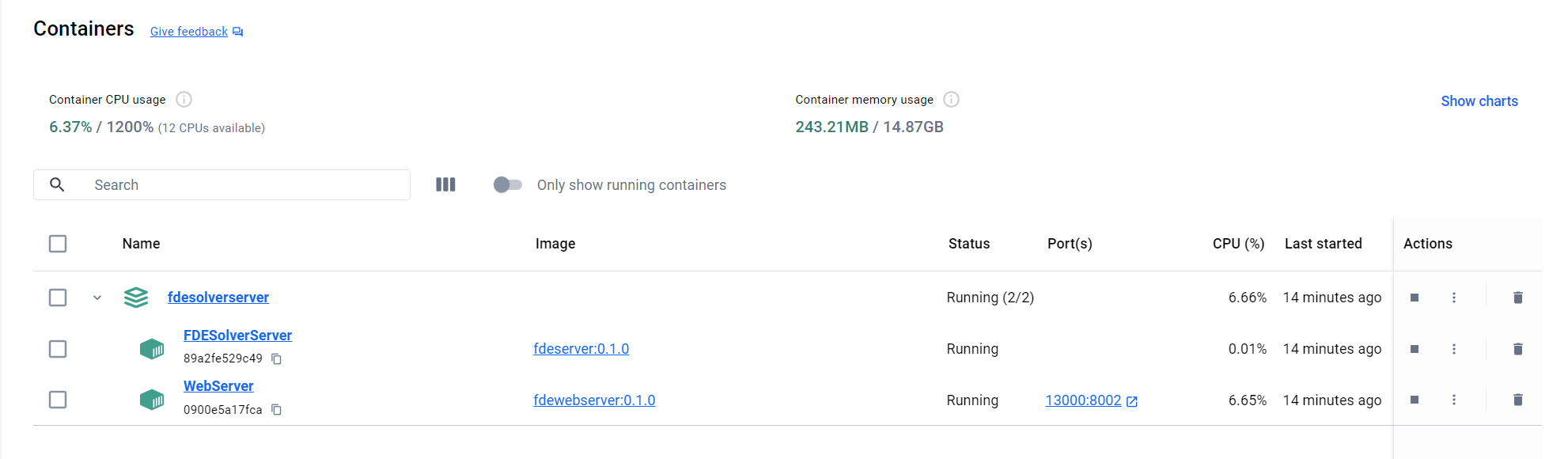


В данном файле берется тот же базовый образ. Однако так как C++ компилируемый язык, то компиляцию следует произвести в контейнере (хотя в теории можно обойтись и без этого). Для этого скачиваются необходимые инструменты: Python для пакетного менеджера Conan, Cmake и компилятор для сборки проекта на C++, Java для генерации файлов ANTLR. Далее проект собирается, после чего сервер запускается.

Файл docker-compose.yml:



Здесь описывается наше приложение, состоящее из двух контейнеров: WebServer и FDESolverServer, находящихся в одной сети client-server типа bridge. После команды docker compose build && docker compose up мы можем увидеть следующее:



**Лабораторная работа 12. Использование ChatGPT для программировани**я **прикладных задач**.

Плюсы использования ChatGPT для программирования прикладных задач:

* **Возможность задавать открытые вопросы**. ChatGPT может стать хорошим источником предложений по структуре приложения. Он также может выдать несколько примеров кода, чтобы дать представление о том, как именно можно приступить к реализации приложения.
* **Более глубокое объяснение того, что делает код**. В отличие от других инструментов, ChatGPT может подробно описать, как работает сгенерированный им код. Это полезно, если работа над приложением только начинается и есть нужда в высокоуровневом руководстве по его реализации и развёртыванию.

Минусы использования ChatGPT для программирования прикладных задач:

* **Ограниченность знаний**. ChatGPT может использовать только те знания, которые были предоставлены ему в качестве обучающей выборки, поэтому он не может дать ответы на вопросы, для которых у него нет достаточных знаний.
* **Ошибки и неточности**. Иногда ChatGPT может генерировать ответы, которые не являются точными или не полностью соответствуют вопросу, что может привести к недопониманию и недовольству пользователей.
* **Необходимость постоянного обучения**. ChatGPT нуждается в постоянном обучении, чтобы сохранять актуальность и точность ответов, что может потребовать дополнительных затрат на время и ресурсы.

Некоторые сценарии и задачи, где ChatGPT помогает автоматизировать процесс разработки:

* **Дизайн пользовательского опыта**. ChatGPT может предоставлять идеи о пути пользователя, макете и дизайне приложения, которые будут удобны.
* **Генерация кода**. Пользователь может разбить большую проблему на более мелкие подзадачи и с помощью ChatGPT получить необходимую помощь. Помимо предоставления кода, ChatGPT также может объяснить, как работает этот код.
* **Поиск ошибок**. Скопировав фрагмент кода и спросив ChatGPT, содержит ли он какие-либо ошибки, можно получить отзыв о качестве кода и выявить потенциальные проблемы. Помимо этого ChatGPT может выявить проблему компиляции и более понятным языком объяснить, как решать проблему.
* **Генерация документации**. Разработчики могут описывать архитектуру приложения, API или конкретные функции, а ChatGPT может создавать информативную документацию, UML диаграмму или примеры кода.
* **Тестирование и обеспечение качества**. ChatGPT может помочь в тестировании приложений, имитируя взаимодействие с пользователем и создавая тестовые сценарии.
* **Непрерывная интеграция и развёртывание**. ChatGPT может предоставить рекомендации по методам и стратегиям CI/CD.
* **Поддержка и помощь пользователям приложений**. ChatGPT может отвечать на часто задаваемые вопросы, давать рекомендации по устранению неполадок или предлагать общую информацию о функциях приложения.

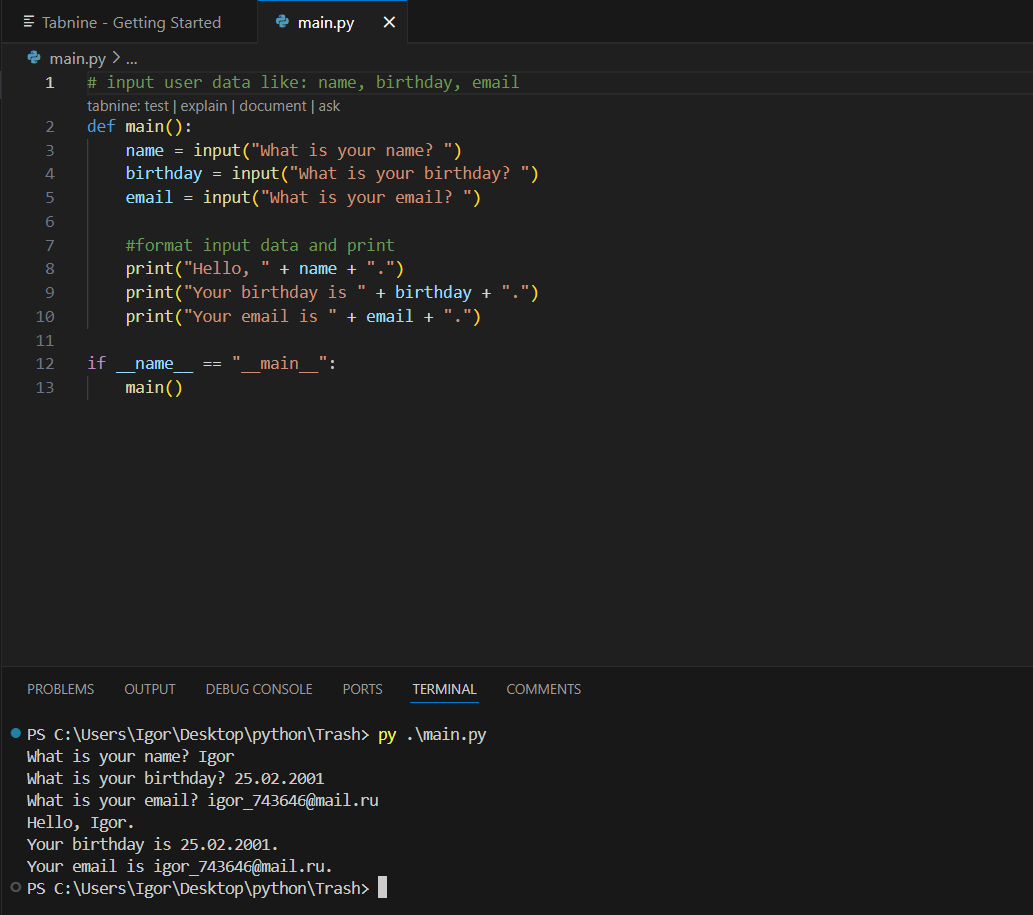
Для обеспечения безопасности кода, генерируемого ChatGPT необходимо его обязательно проверять на:

* Соответствие последним безопасным версиям используемых инструментов.
* Типичные ошибки безопасности используемого языка программирования (выход за границы памяти, использование неинициализированных переменных, взаимозависимость времени жизни объектов, недопущение взаимоблокировок потоков и т.д.).
* Использование наилучших практик по защите пользовательских данных и данных доступа.

Для наилучшего пользовательского опыта при разработе с ChatGPT следует пользоваться следующими этическими требованиями:

* Устранение предвзятости системы ИИ, которые могут непреднамеренно увековечить предубеждения, присутствующие в ответах.
* Чрезмерная зависимость от ИИ в общении может привести к снижению навыков межличностного общения или зависимости от технологий. Важно найти баланс между взаимодействием с помощью ИИ и человеческим общением/обучением.

Пример сгенерированного кода Tabnine по комментариям:



Программа, которая считывает вопросы из файла и обращается к ChatGPT, чтобы получить ответы на вопросы:

