**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

«**Сибирский государственный университет науки и технологий имени академика М.Ф. Решетнева»**

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

**КУРСОВАЯ РАБОТА**

по дисциплине: «Языки программирования»

на тему: «Разработка редактора кода с подсветкой синтаксиса»

Выполнил студент группы *БПИ18-01*

*очной* формы обучения

*Рылов А.Е*

Дата сдачи: «\_\_\_\_»\_\_\_\_\_\_\_\_2019 г.

Дата защиты: «\_\_\_\_»\_\_\_\_\_\_\_\_2019 г.

Оценка:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись руководителя)

Красноярск 2019 г.

**ЗАДАНИЕ**

на курсовую работу по дисциплине *Языки программирования*

студентом *Рылов А.Е.* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Группа *БПИ18-01* Форма обучения *Очная*

1. Тема работы (проекта):\_\_Разработка редактора кода с подсветкой синтаксиса

2. Срок сдачи студентом работы

3. Перечень вопросов, подлежащих разработке при написании теоретической части:

1) Анализ функционала популярных редакторов кода \_\_\_\_\_\_ 2) Выбор языка и технологии разработки 3) Разбор реализации функций 4) Выбор IDE для разработки редактора кода

4. Перечень вопросов, подлежащих разработке при написании практической части:

1) Описание работы подсветки синтаксиса 2) Описание создания графической оболочки и нумерации строк 3) Компиляция приложения из исходного кода

5. Дата выдачи задания:

Руководитель: Богданов К. В.

(подпись руководителя)

Задание принял к исполнению (дата):

(подпись студента)

СОДЕРЖАНИЕ

[1 ВВЕДЕНИЕ 4](#_Toc10387628)

[2 АНАЛИЗ ОБЛАСТИ СОЗДАНИЯ РЕДАКТОРОВ КОДА 6](#_Toc10387629)

[2.1 Анализ функционала популярных редакторов кода. 6](#_Toc10387630)

[2.2 Выбор языка и технологии разработки 10](#_Toc10387631)

[2.3 Реализация выбранных функций редактора кода 13](#_Toc10387632)

[2.4 Выбор IDE для разработки редактора кода 13](#_Toc10387633)

[3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ 14](#_Toc10387634)

[3.1 Работа подсветки синтаксиса (PythonHighlighter.py) 15](#_Toc10387635)

[3.2 Описание графической оболочки и нумерации строк (UI.py) 16](#_Toc10387636)

[3.2.1 Описание создания нумерации строк 16](#_Toc10387637)

[3.2.2 Создание основного UI 17](#_Toc10387638)

[3.3 Объединение всего (main.py) 18](#_Toc10387639)

[3.4 Компиляция приложения 19](#_Toc10387640)

[3.5 Руководство пользователя 21](#_Toc10387641)

[4 ЗАКЛЮЧЕНИЕ 24](#_Toc10387642)

[ИСПОЛЬЗОВАННЫЕ РЕСУРСЫ 25](#_Toc10387643)

### 1 ВВЕДЕНИЕ

*Актуальность.* Уже все и забыли времена, когда программисты писали свой код (программы) в обычных блокнотах, без подсветки синтаксиса, без автодополнения и прочих разных помощников.

Сейчас все, за очень маленьким исключением, программисты используют разные редакторы кода и IDE, вроде Sublime Text 3, Visual Studio, Atom, Xcode, PyCharm.

Все эти редакторы кода и IDE очень разные, и каждый сможет найти подходящий именно ему. По цвету подсветки синтаксиса, по цвету бэкграунда, то есть по цветовому оформлению, а также по идеологии.

В настоящее время на GitHub и подобных ему сервисах появляются новые редакторы кода, от независимых разработчиков, энтузиастов, а так же разные компании пытаются выпустить свою версию, лучшую, по их мнению, чем другие и выделяющиеся чем-то новым, вроде интеграция с другими сервисами и тому подобное.

Мы тоже решили сделать свой редактор кода. Конечно, он пока не выделяется чем-то от других. Даже больше – у него нет некоторых функций, которые уже давно стали базовыми в редакторах кода. Но эта наработка, база в будущем может стать фундаментом для чего-то большего.

*Цель работы.* Целью курсовой работы является разработка кроссплатформенного редактора кода, с самым базовым функционалом.

Для достижения поставленной цели необходимо решить следующие задачи:

- провести анализ функций популярных редакторов кода

- выбрать язык для разработки

- выбрать технологию (фреймворк) и разораться в нужных модулях

- выбрать IDE для разработки

- разработать графический интерфейс пользователя (GUI)

- разработать подсветку синтаксиса

- разработать другие функции

- собрать код в приложения для запуска на любом компьютере

### 2 АНАЛИЗ ОБЛАСТИ СОЗДАНИЯ РЕДАКТОРОВ КОДА

### 2.1 Анализ функционала популярных редакторов кода.

*Sublime Text (рис.1)*

**

Рисунок 1

Он довольно-таки давно обосновался в топах редакторов кода. Sublime Text разработал бывший инженер Google, Джон Скиннер. В Sublime Text очень много удобных функций, помогающих программистам писать свой код. Он платен (около 70$), но его можно использовать и бесплатно, но с постоянными напоминаниями, что вы не заплатили им. А также он кроссплатформенный.

- Подсветка синтаксиса

- Настройка шрифтов

- Настройка общей темы оформления

- Автоматическое проставление табуляций во вложенных конструкциях

- Нумерация строк и положение курсора

- Возможность открывать несколько вкладок

- Возможность выбрать кол-во пробелов в tab

- Превью код в правом углу редактора (у него одного)

- Возможность написания своих плагинов на Python

- Поиск по коду, замена

- Все, чего в нем нет, возможно установить плагинами

Sublime Text очень быстрый и гладкий

*Atom (рис.2)*

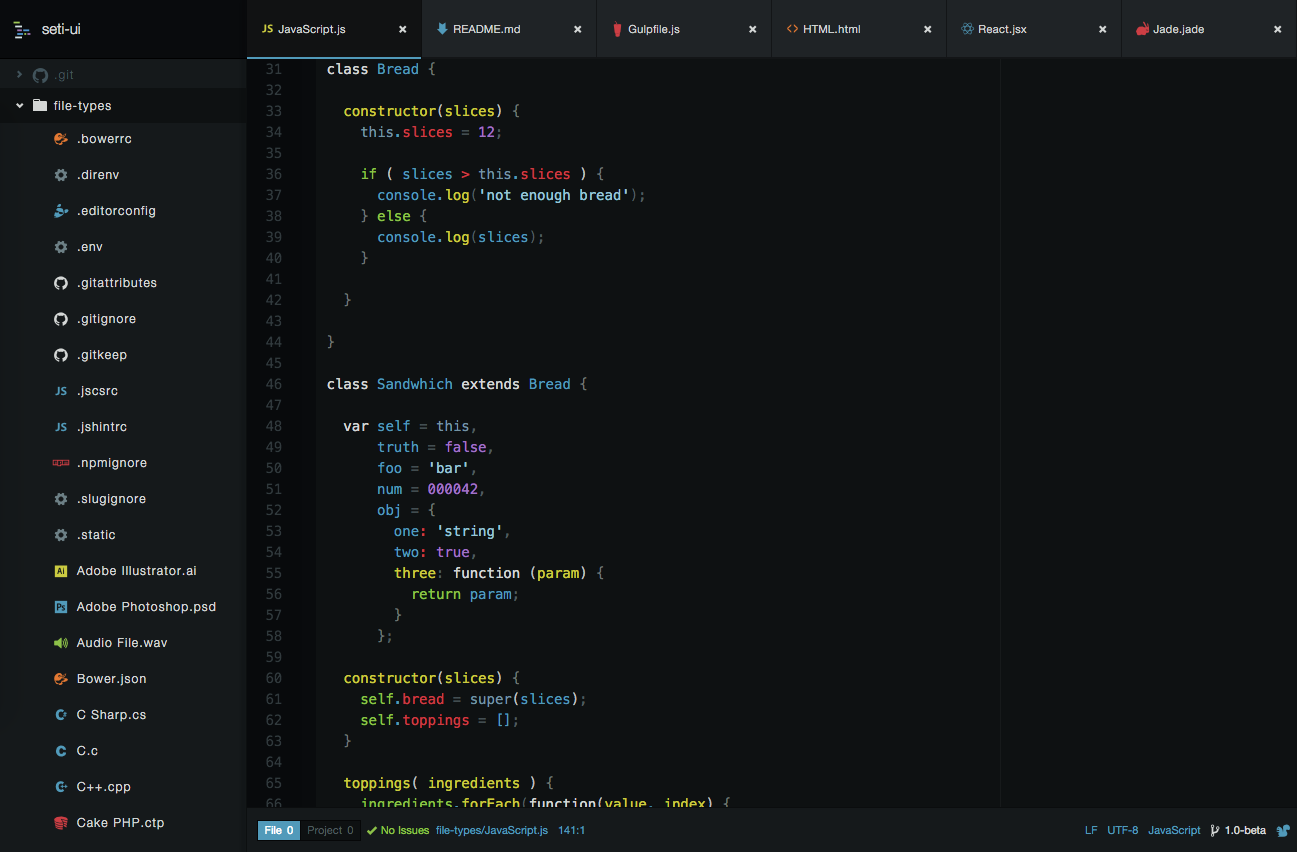
**

Рисунок 2

Atom – относительный новичок в мире текстовых редакторов, но получил огромный импульс с тех пор, как был впервые выпущен в 2014 году. Разработчиком Atom является GitHub, что позволяет ему очень хорошо работать с Git и тому подобное. Тоже кроссплатформенный.

- Подсветка синтаксиса

- Настройка общей темы оформления (очень гибкая)

- Автоматическое проставление tab во вложенных конструкциях

- Нумерация строк и положение курсора

- Возможность открытия нескольких вкладок

- Возможность выбрать кол-во пробелов в tab

- Возможность написания своих плагинов (очень гибко)

- Поиск по коду, замена

- Отлично работает с Git (!)

- И опять же, все чего нет «с коробки» можно установить плагинами

Atom, в отличии от Sublime Text, немного тормозит.

*Visual Studio Code (рис.3)*

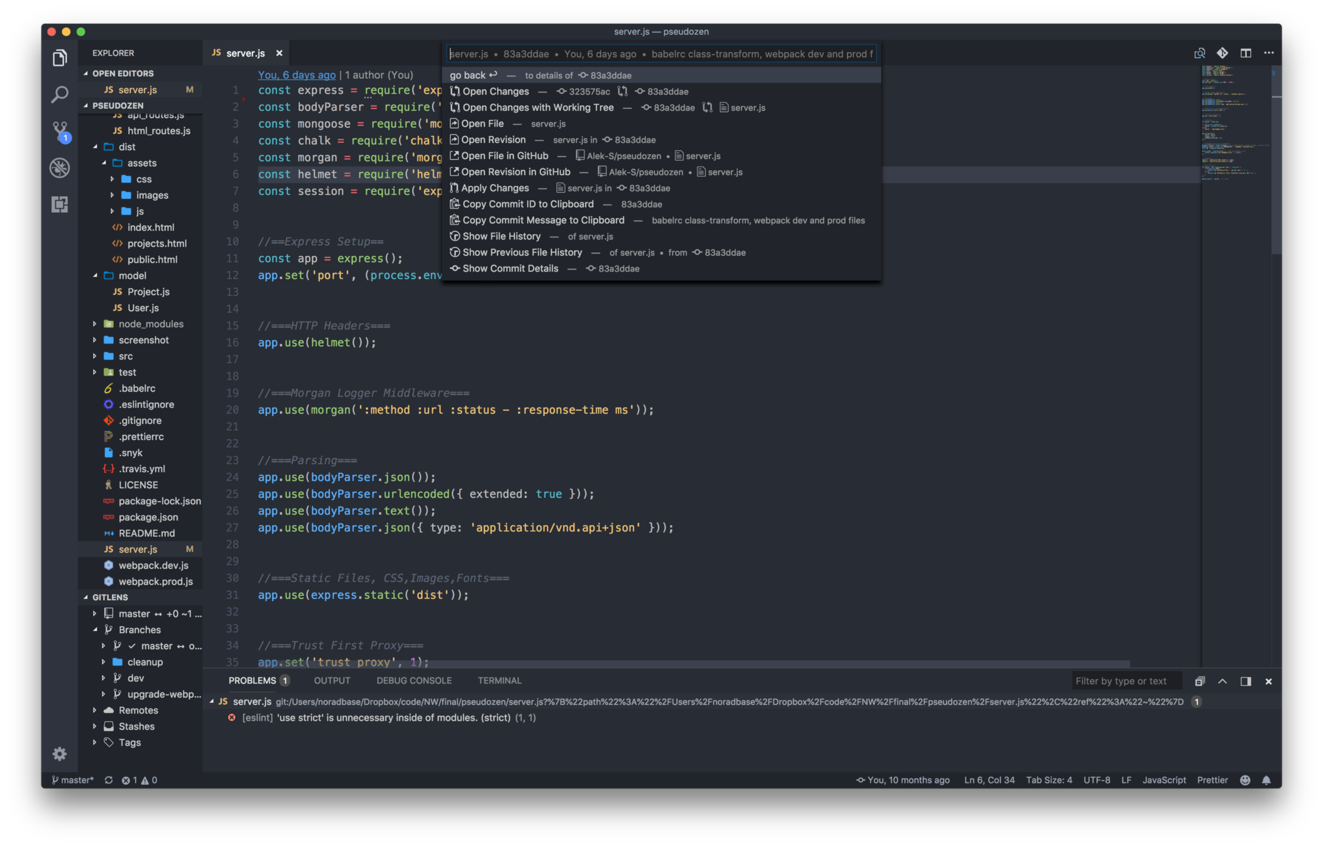
**

Рисунок 3

Этот редактор кода был произведен Microsoft, и распространяется бесплатно. Он появился на свет в 2015 году и до сих пор набирает обороты и завоевывает любовь программистов.

- Большинство функций предыдущих редакторов кода

- Намного удобней происходит установка расширений (плагинов)

- Удобная интеграция с Git

- Имеется отладчик

*Vim (рис.4)*

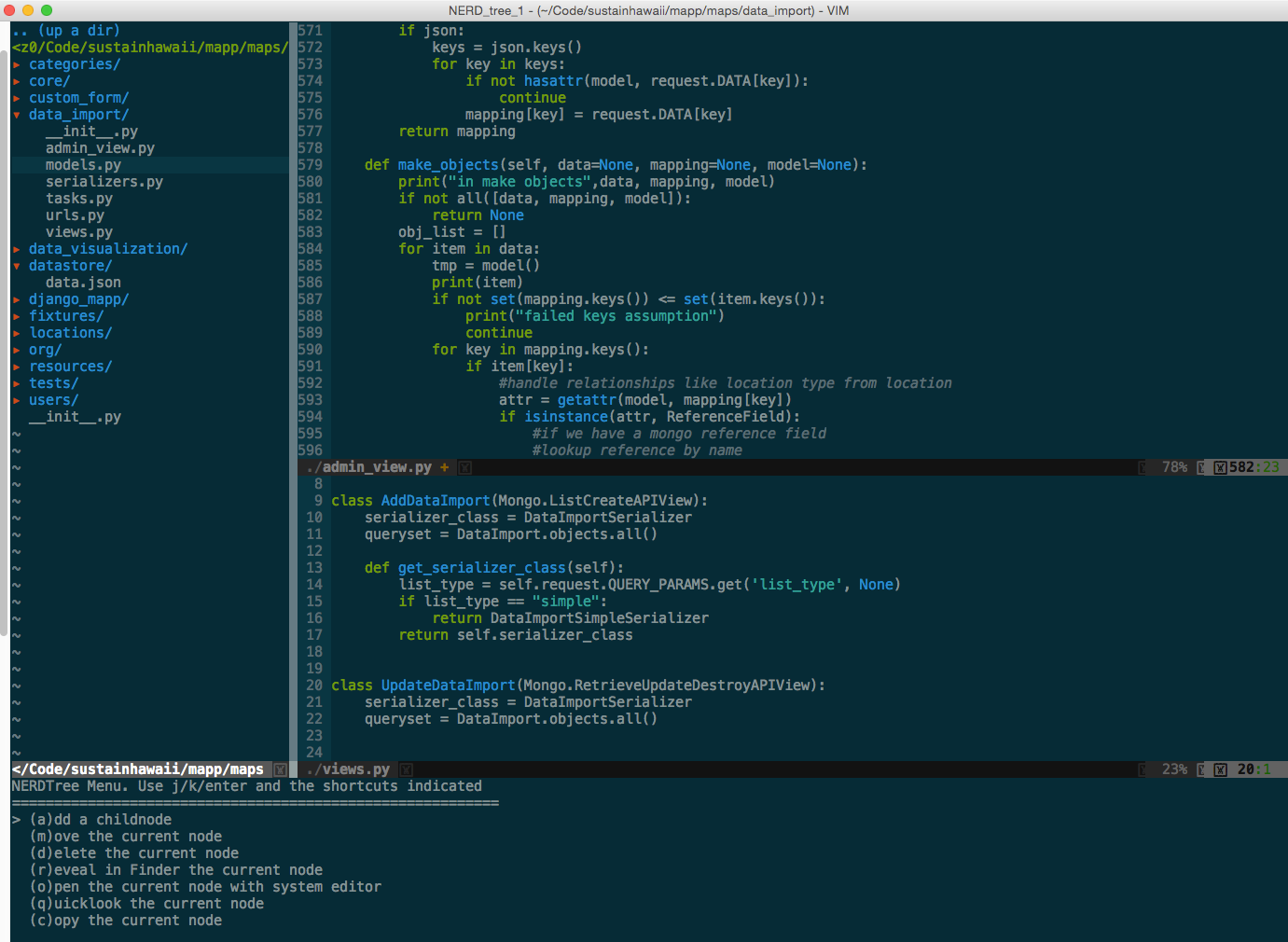
**

Рисунок 4

Vim – это редактор кода. У него есть безгранично много функций, так как к сегодняшнему дню к нему было написано более 14000 пакетов. Единственная его проблема – это проблема его освоения, он очень уж труден для новичков. Но когда программист изучает его – он становится очень продуктивной средой. Распространяется он бесплатно, разработчик – Bram Moolevar. Является кроссплатформенным.

- Большинство функций предыдущих редакторов кода

- Возможность редактирования или просмотра файла на удаленном сервере через терминал (!)

- Плагинов тут больше, чем где либо

- Очень быстрый, все зависит от вас

Можно заметить, что большинство функций различных редакторов кода совпадают, а отличают их лишь некоторые возможности интеграции с разными сервисами, работы через терминал и тому подобное.

Мы нацелимся на реализацию *подсветки синтаксиса, нумерации строк и отображения положения курсора.*

### 2.2 Выбор языка и технологии разработки

Языком для реализации этой задачи был выбран Python. Python – высокоуровневый, объектно-ориентированный язык программирования общего назначения, ориентированный на повышение производительности программиста и улучшение читаемости кода.

Нам нужно написать редактор кода, без графического интерфейса пользователя (GUI) это невозможно. Для Python есть много разных библиотек, которые позволяют создавать GUI, их более 30, но сейчас перечислю 12 самых популярных из них с небольшой характеристикой.

1. *Tinter*

Tinter – это набор инструментов, который может формировать GUI с помощью Python. Он позволяет вам запускать сценарии на Python в GUI-формате.

2. *Flexx*

Многие GUI-библиотеки Python основаны на библиотеках, написанных на других языках, таких как C++. Например, wxWidgets и libavg. Flexx создана на Python. Для отображения GUI используют веб-технологию.

3. *CEF Python*

Этот фреймворк основан на Google Chromium. Основное внимание в нем уделяется облегчения использования встроенного браузера в сторонних приложениях

4. *Dabo*

Цель этого фреймворка – WxPython. Это трехуровневый фреймворк для разработки кросс-платформенных приложений.

5. *Kivv*

Он основан на OpenGL ES 2. У него есть собственный multi-touch для каждой отдельной платформы, Это событийно-ориентированный фреймворк, который прекрасно подходит для разработки игр.

6. *Pyforms*

Pyforms – это фреймворк Python для разных окружений, используемый для разработки GUI приложения. В нем поощряется повторное использование кода.

7. *PyGObject*

С помощью PyGObject вы можете писать на Python приложения для проекта GNOME. Так же можно писать приложения на Python, использующие GTK+.

8. *PyQt*

Qt – это кросс-платформенный фреймворк. Он написан на C++. Это очень полная библиотека, включающая много инструментов и API. Широко используется во многих отраслях. Охватывает множество платформ.

9. *PySide*

Обертка для Qt. PySide и Qt отличаются тем, что последний является коммерчески доступным.

10. *PyGUI*

Фокус этого MVC-фреймворка заключается в том, чтобы вписываться в экосистему Python с максимальной легкостью.

11. *libavg*

Это сторонняя библиотека, написанная на C++. Обладает следующими особенностями:

- показ элементов в форме переменных Python

- система управления событиями

- таймеры

- поддержка логов

12. *wxPython*

Обертка кросспатформенного GUI для Python на основе wxWidgets. Написана на C++.

Сравнение претендентов

|  |  |  |  |
| --- | --- | --- | --- |
| **Характеристика** | **Библиотека** | | |
| **Tinter** | **PyQt** | **PySide** |
| Хорошая документация | Нет | Да | Да |
| Бесплатна | Да | Да | Нет |
| Наличие уроков | Да | Да | Нет |
| Простота использования | Да | Да | Да |
| Интеграция с несколькими языками программирования | Нет | Да | Да |
| Наличие нужных методов | Нет | Да | Да |

После анализа всех фреймворков для создания GUI, мы выбрали PyQt, а если быть точнее – PyQt5.

Это библиотека содержит нужные нам методы форматирования текста, организации виджетов на окне и тому подобное.

На этом фреймворке уже существует несколько написанных редакторов кода, а это уже о чем-то да говорит.

### 2.3 Реализация выбранных функций редактора кода

Функции, выбранные для реализации в работе:

- Подсветка синтаксиса

- Нумерация строк

- Отображение положения курсора

Реализация *отображения положения курсора.*

Нам нужно, чтобы показания положения курсора всегда изменялись, когда изменяется положение курсора в нашей рабочей области, в редакторе. В PyQt5 уже есть методы, которые отслеживают изменение положения курсора, которыми я буду пользоваться.

Реализация *подсветки синтаксиса*.

Будем искать нужные нам слова и символы (ключевые слова, операторы, скобки, наименование классов и функций, однострочные комментарии, многострочные комментарии, числа) с помощью построенных нами регулярных выражений. В PyQt5 имеются нужные средства, для поиска с помощью регулярных выражений и нужные к этому методы. Искать и подсвечивать нужно каждый раз, когда на поле редактирования происходят изменения.

Реализация *нумерация строк.*

Чтобы нумерация строк работала корректно – нужно, чтобы «рисовалась» новая строчка с следующим числом в отдельном виждете слева. Так же нужно по необходимости расширять этот виджет, чтобы номера любой длинны помещались в него. PyQt5 опять содержит все нужные нам методы.

### 2.4 Выбор IDE для разработки редактора кода

Так как для разработки редактора кода был выбран язык программирования Python, то разработка будет осуществляться в IDE от JetBrains PyCharm.

### 3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

Вид приложения, к которому мы будем стремиться: (рис.5)

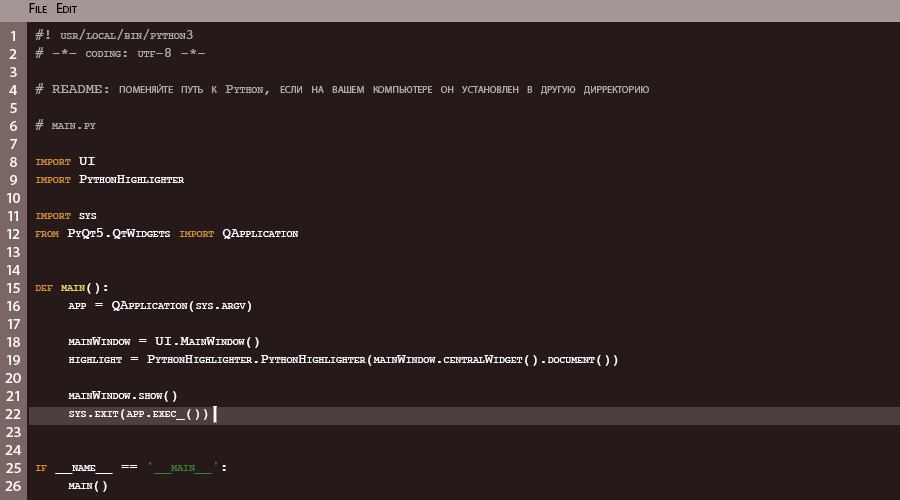


Рисунок 5

Программа будет разделена на 3 модуля:

- *PythonHighlighter.py*

В этом модуле опишем работу подсветки синтаксиса

- *UI.py*

В этом модуле создадим графический интерфейс и нумерация строк

- *main.py*

В этому модуле объединяем все и заставляем работать вместе

UML-диаграмма классов: (рис.6)

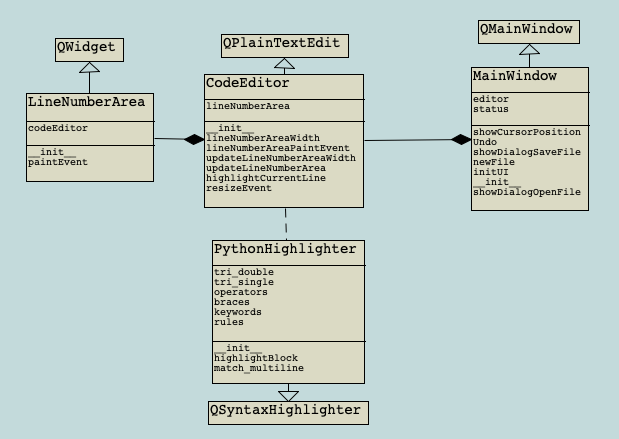


Рисунок 6

### 3.1 Работа подсветки синтаксиса (PythonHighlighter.py)

Код модуля в «приложение 1»

На строках:

**3-4** подключаем нужные модули

**6-18** Функция format, к которой мы обращаемся при формировании стилей. QTextCharFormat отвечает за редактирование символов.

**22-32** Формируем стили, которые будут применяться к разным группам слов

**35-181** Описание класса для подсветки синтаксиса:

**39-46** Создаем список из всех ключевых слов Python

**49-59** Создаем список всех операторов Python

**61-64** Создаем список всех скобок Python

**66-113** Описываем \_\_init\_\_:

**67** Передаем document в QSyntaxHighlighter для корректной работы подсветки

**70-71** Формируем регулярные правила для подсветки многострочных коментариев

**73-82** Формируем регулярные правила для подсветки ключевых слов, операторов, скобок

**84-109** Формируем правила для идентификаторов функций и классов и т. д.

**112-113** Записываем все правила в тип QRegExp и делаем из списка атрибут класса

**115-138** Переопределяем highlightBlock из наследуемого класса QSyntaxHighlighter, эта функция вызывается всегда, когда просходит какое-либо изменение блока:

**124-130** Описываем цикл, который будет «ходить» по строке и подсвечивать встретившиеся слова, подходящие под правила

**133** Устанавливаем в currentBlockStart 0, что говорит о том, что мы не находимся в многострочном комментарии

**136-138** Подсвечиваем многострочные комментарии, вызывая функцию match\_multiline

**140-181** Описание работы подсветки многострочных комментариев:

**150-157** Проверяем, находимся ли мы в многострочном комментарии или нет

**160-174** Подсвечиваем строки в цвет многострочного комментария пока не увидим ограничитель в виде трех двойных/одинарных кавычек

**178-181** Возвращаем True, если до сих пор в многострочном комментарии, иначе - False

### 3.2 Описание графической оболочки и нумерации строк (UI.py)

### 3.2.1 Описание создания нумерации строк

Код модуля в «приложение 2»

**3-5** Подключаем все нужные модули

**9-15** Создаем класс «рисовки» номеров строк

**18-100** Создаем класс поля для нашего редактора кода, наследуя его от QPainTextEdit, который позволяет писать и редактировать текст в поле с нужными методами:

**19-31** Описываем \_\_init\_\_:

**22** Создаем экземпляр класса LineNumberArea, который был создан нами

**24** При изменении редактируемого блока будет происходить дополнение ширины виджета с номерами (по необходимости)

**27** При изменении строки будет рисовать новое число в виждете с номерами строк

**30** При изменении позиции курсора будет подсвечиваться текущая строки

**33-42** Описываем функцию, которая будут возвращать нужную нам длину виджета для нумерации

**44-54** Функция, добавляющая ширину виждета, и функция, добавляющая новую строку в виждете

**56-60** Функция, которая переназначает размер виждета с нумерацией строк, мы переопределили его из класса QPlainTextEdit

**62-76** Описываем функция, подсвечивающую текущую редактируемую строку

**78-100** Описываем функцию, которая рисует новые номера с помощью QPainter

### 3.2.2 Создание основного UI

**103-215** Создание класса MainWindow, который наследуется от QMainWindow, содержащий методы для создания полноценного графического окружения:

**104-106** Описание \_\_init\_\_

**108-168** Описание initUI:

**113** Создаем экземпляр класса CodeEditor, который был создан нами

**114-119** Задаем стили к редактируемому полю

**120** Устанавливаем ширину одного tab

**125-126** Устанавливаем поле для редактирования центральным виждетом главного окна, задаем ему размер 900х500 пикселей.

**131-161** Создание меню бара. Добавляем действие, наздачаем горячие клавиши, и подключаем функцию, которая будет исполнять. Затем добавляем все действия в меню File и Edit.

**167-168** Создание статус бара, в котором будет отображаться положение курсора

**173-181** Описание работы функции открытия файла

**184-192** Описание работы функции сохранения файла

**195-202** Описание работы функции создания нового файла

**207-211** Описание работы функции отображения курсора в статус баре

**214-215** Функция отмены последнего действия

### 3.3 Объединение всего (main.py)

Код модуля в «приложение 3»

**9-13** Подключаем модули для запуска программы.

**17** Создаем приложение

**19** Создаем главное окно

**20** Добавляем подсветку в центральный виджет главного окна (поле для редактирования)

**23-24** Показываем главное окно и входим в бесконечный цикл приложения.

### 3.4 Компиляция приложения

Для компиляции приложения был выбран pyinstaller.

Чтобы скомпилировать приложение нужно набрать такую команду в терминале, находясь в директория наших файлов:

pyinstaller --onefile --noconsole main.py

Эта команда скомпилирует приложение в один файла с запуском без консоли.

Получившаяся папка с приложением (рис.7)



Рисунок 7

dist хранит в себе сами приложения (рис.8)

****

Рисунок 8

Build/main хранит все нужные модули, для работы программы (рис.9)

****

Рисунок 9

### 3.5 Руководство пользователя

Мы максимально приблизились к предполагаемому виду приложения с рис.5

Так выглядит начальный экран: (рис.10)



Рисунок 10

Меню: (рис.11)

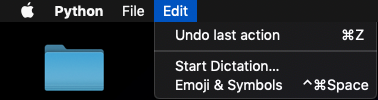
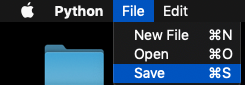


Рисунок 11

Окно открытия файла: (рис.12)

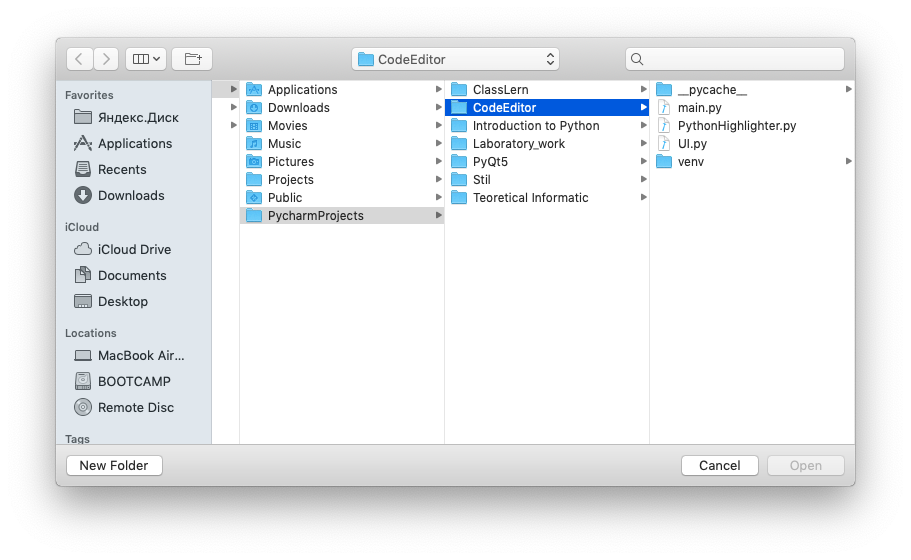


Рисунок 12

Окно сохранения файла: (рис.13)

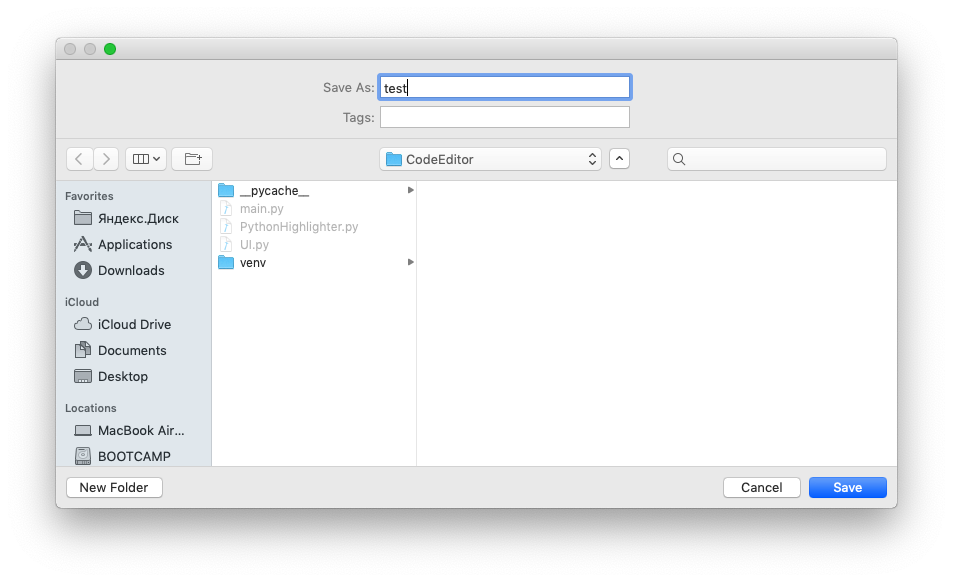


Рисунок 13

Демонстрация подсветки синтаксиса: (рис.14)

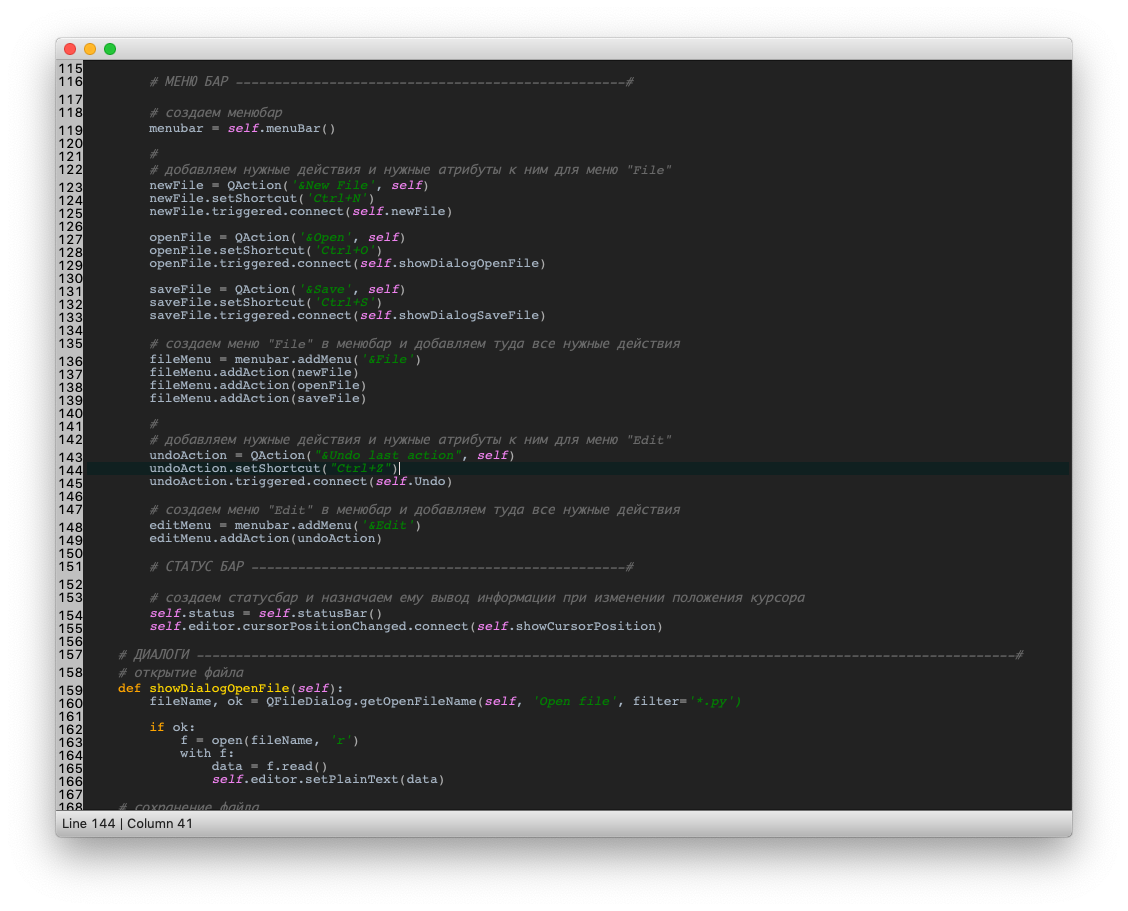


Рисунок 14

### 4 ЗАКЛЮЧЕНИ

В результате выполнения курсовой работы был разработан редктор кода с подсветкой кода и другими функциями. Мы максимально приблизились к предполагаемому виду приложения, спроектированного в начале

В ходе анализа решаемой задачи был сделан обзор на функционал популярных редакторов кода, обзор на библиотеки, позволяющие создавать GUI.

Был выбран язык программирования, IDE и технологию, с помощью которой будет решена данная задача.

Описали работу всех редакторов кода. А так же описали создание редактора кода с использованием Python и PyQt5 на примере своего редактора кода.

Скомпилировали приложение из исходного кода с помощью pyinstaller.

Сделали обзор на получившееся приложения и продемонстрировали его возможности.

### ИСПОЛЬЗОВАННЫЕ РЕСУРСЫ

Сделали обзор на получившееся приложения и продемонстрировали его возможности.

1) О Sublime Text

[**https://eax.me/sublime-text/**](https://eax.me/sublime-text/)

2) Сравнение разных редакторов кода

[**https://mediadoma.com/atom-obzor-sravnenie-s-sublime-visual-studio-code-vim**](https://mediadoma.com/atom-obzor-sravnenie-s-sublime-visual-studio-code-vim)

3) 13 GUI-библиотек

[**https://techrocks.ru/2018/04/26/13-python-gui-frameworks/**](https://techrocks.ru/2018/04/26/13-python-gui-frameworks/)

4) Документация Qt

[**https://doc.qt.io**](https://doc.qt.io)

5) Обучение PyQt5

[**https://pythonworld.ru/gui/pyqt5-firstprograms.html**](https://pythonworld.ru/gui/pyqt5-firstprograms.html)

**Приложение 1**

# PythonHighlighter.py

from PyQt5.QtCore import QRegExp

from PyQt5.QtGui import QColor, QTextCharFormat, QSyntaxHighlighter

def format(color, style=''):

*"""Возвращает QTextCharFormat с данными атрибутами."""*

\_color = QColor()

\_color.setNamedColor(color)

\_format = QTextCharFormat()

\_format.setForeground(\_color)

if 'italic' in style:

\_format.setFontItalic(True)

return \_format

# Стили, приминимые к разнам группам слов

STYLES = {

'keyword': format('orange'),

'operator': format('darkgray'),

'brace': format('darkGray'),

'def\_class': format('gold'),

'string': format('green', 'italic'),

'comment1': format('dimgray', 'italic'),

'comment2': format('greenyellow'),

'self': format('violet', 'italic'),

'numbers': format('brown'),

}

class PythonHighlighter (QSyntaxHighlighter):

*"""Подсветчик синтаксиса для Python."""*

# Ключевые слова Python

keywords = [

'and', 'assert', 'break', 'class', 'continue', 'def',

'del', 'elif', 'else', 'except', 'exec', 'finally',

'for', 'from', 'global', 'if', 'import', 'in',

'is', 'lambda', 'not', 'or', 'pass', 'print',

'raise', 'return', 'try', 'while', 'yield',

'None', 'True', 'False'

]

# Операторы Python

operators = [

'=',

# Сравнение

'==', '!=', '<', '<=', '>', '>=',

# Арифметика

'\+', '-', '\\*', '/', '//', '\%', '\\*\\*',

# Сокращенная арифметика

'\+=', '-=', '\\*=', '/=', '\%=',

# Битовые операторы

'\^', '\|', '\&', '\~', '>>', '<<',

]

# Скобки Python

braces = [

'\{', '\}', '\(', '\)', '\[', '\]',

]

def \_\_init\_\_(self, document):

QSyntaxHighlighter.\_\_init\_\_(self, document)

# Многострочные коментарии (регулярное выражение, флаг, стиль)

self.tri\_single = (QRegExp("'''"), 1, STYLES['comment2'])

self.tri\_double = (QRegExp('"""'), 2, STYLES['comment2'])

rules = []

# (регулярное выражение, nth, стиль)

# Добавляем правила для ключевых слов, операторов, скобок

rules += [(r'\b%s\b' % w, 0, STYLES['keyword'])

for w in PythonHighlighter.keywords]

rules += [(r'%s' % o, 0, STYLES['operator'])

for o in PythonHighlighter.operators]

rules += [(r'%s' % b, 0, STYLES['brace'])

for b in PythonHighlighter.braces]

# Добавляем другие правила

rules += [

# 'self'

(r'\bself\b', 0, STYLES['self']),

# Строки в двойных кавычках, которые могут содержать escape-последовательности

(r'"[^"\\]\*(\\.[^"\\]\*)\*"', 0, STYLES['string']),

# Строки в одинарных кавычках, которые могут содержать escape-последовательности

(r"'[^'\\]\*(\\.[^'\\]\*)\*'", 0, STYLES['string']),

# Идентефикатор def'а

(r'\bdef\b\s\*(\w+)', 1, STYLES['def\_class']),

# Идентефикатор class'а

(r'\bclass\b\s\*(\w+)', 1, STYLES['def\_class']),

# Коментарий от # до новой строки

(r'#[^\n]\*', 0, STYLES['comment1']),

# Числа

(r'\b[+-]?[0-9]+[lL]?\b', 0, STYLES['numbers']),

(r'\b[+-]?0[xX][0-9A-Fa-f]+[lL]?\b', 0, STYLES['numbers']),

(r'\b[+-]?[0-9]+(?:\.[0-9]+)?(?:[eE][+-]?[0-9]+)?\b', 0, STYLES['numbers']),

]

# Создаем QRegExp для каждого шаблона

self.rules = [(QRegExp(pat), index, fmt)

for (pat, index, fmt) in rules]

def highlightBlock(self, text):

*"""Применяет подсветку синтаксиса следуя правилам, описанных нами.*

*Вызывается, когда в тексте меняются блоки текста.*

*(Мы переопределяем эту функцию из QSyntaxHighlighter)"""*

# Подсвечиваем синтаксис (expression - регулярное выражение для поиска

# nth - n-ое совпадение нужное нам

# format - стили редактируемого синтаксиса)

for expression, nth, format in self.rules:

index = expression.indexIn(text, 0)

length = len(expression.cap(nth))

while index >= 0:

index = expression.pos(nth)

self.setFormat(index, length, format)

index = expression.indexIn(text, index + length)

# Нужно для подсветки многострочных коментариев

self.setCurrentBlockState(0)

# Подсвечиваем многострочные коментарии

in\_multiline = self.match\_multiline(text, \*self.tri\_single)

if not in\_multiline:

in\_multiline = self.match\_multiline(text, \*self.tri\_double)

def match\_multiline(self, text, delimiter, in\_state, format):

*"""Подсвечиваем многострочные коментарии.*

*'delimiter' - это ограничение в виде QRegExp для тройных одинарных или двойных кавычек,*

*'in\_state' - уникальное целое число для представления соответствующих изменений состояния внутри этих строк,*

*'format' - стили, применимые к многострочным коментариям.*

*Возвращает True, если мы до сих пор в многострочном коментарии."""*

# Если внутри тройных одинарных кавычек, то начать сначала отсчет

if self.previousBlockState() == in\_state:

start = 0

add = 0

# В другом случае - искать ограничение (кавычки) на этой строке

else:

start = delimiter.indexIn(text)

# Длинна совпадения

add = delimiter.matchedLength()

# Пока на этой линии есть ограничитель (кавычки)

while start >= 0:

# Находит завершиющий ограничитель (кавычки)

end = delimiter.indexIn(text, start + add)

# Зарершающий ограничитель (кавычки) на этой строке?

if end >= add:

length = end - start + add + delimiter.matchedLength()

self.setCurrentBlockState(0)

# Нет? значит все еще в многострочном комментарии

else:

self.setCurrentBlockState(in\_state)

length = len(text) - start + add

# Добавляем стили

self.setFormat(start, length, format)

# Смотрим ищем следующий ограничитель

start = delimiter.indexIn(text, start + length)

# Возвращаем True, если до сих пор в многострочном коментарии, иначе - False

if self.currentBlockState() == in\_state:

return True

else:

return False

**Приложение 2**

# UI.py

from PyQt5.QtWidgets import (QMainWindow, QAction, QFileDialog, QMessageBox, QWidget, QPlainTextEdit, QTextEdit)

from PyQt5.QtCore import Qt, QRect

from PyQt5.QtGui import QColor, QPainter, QTextFormat

class LineNumberArea(QWidget):

def \_\_init\_\_(self, editor):

super().\_\_init\_\_(editor)

self.codeEditor = editor

def paintEvent(self, event):

self.codeEditor.lineNumberAreaPaintEvent(event)

class CodeEditor(QPlainTextEdit):

def \_\_init\_\_(self):

super().\_\_init\_\_()

# создаем экземпляр класса LineNumberArea

self.lineNumberArea = LineNumberArea(self)

# изменение, дополнение ширины виджета с номерами строк

self.blockCountChanged.connect(self.updateLineNumberAreaWidth)

# новая строка с нумерацией будет "рисоваться" каждый раз, когда будет меняться форм. блок

self.updateRequest.connect(self.updateLineNumberArea)

# выделение (подсвечивание) редкатируемой строки меняется когда менятся позиция курсора

self.cursorPositionChanged.connect(self.highlightCurrentLine)

self.updateLineNumberAreaWidth(0)

def lineNumberAreaWidth(self):

digits = 1

max\_value = max(1, self.blockCount())

while max\_value >= 10:

max\_value /= 10

digits += 1

space = 3 + self.fontMetrics().width(' ') \* digits

return space

def updateLineNumberAreaWidth(self, \_):

self.setViewportMargins(self.lineNumberAreaWidth(), 0, 0, 0)

def updateLineNumberArea(self, rect, dy):

if dy:

self.lineNumberArea.scroll(0, dy)

else:

self.lineNumberArea.update(0, rect.y(), self.lineNumberArea.width(), rect.height())

if rect.contains(self.viewport().rect()):

self.updateLineNumberAreaWidth(0)

def resizeEvent(self, event):

super().resizeEvent(event)

cr = self.contentsRect()

self.lineNumberArea.setGeometry(QRect(cr.left(), cr.top(), self.lineNumberAreaWidth(), cr.height()))

def highlightCurrentLine(self):

extraSelections = []

if not self.isReadOnly():

selection = QTextEdit.ExtraSelection()

lineColor = QColor()

lineColor.setNamedColor('#102020')

lineColor.lighter(500)

selection.format.setBackground(lineColor)

selection.format.setProperty(QTextFormat.FullWidthSelection, True)

selection.cursor = self.textCursor()

selection.cursor.clearSelection()

extraSelections.append(selection)

self.setExtraSelections(extraSelections)

def lineNumberAreaPaintEvent(self, event):

painter = QPainter(self.lineNumberArea)

painter.fillRect(event.rect(), Qt.lightGray)

block = self.firstVisibleBlock()

blockNumber = block.blockNumber()

top = self.blockBoundingGeometry(block).translated(self.contentOffset()).top()

bottom = top + self.blockBoundingRect(block).height()

height = self.fontMetrics().height()

while block.isValid() and (top <= event.rect().bottom()):

if block.isVisible() and (bottom >= event.rect().top()):

number = str(blockNumber + 1)

painter.setPen(Qt.black)

painter.drawText(0, top, self.lineNumberArea.width(), height, Qt.AlignRight, number)

block = block.next()

top = bottom

bottom = top + self.blockBoundingRect(block).height()

blockNumber += 1

class MainWindow (QMainWindow):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

# ПОЛЕ РЕДАКТОРА -------------------------------------------#

# создаем поле для редактирования кода, наздачаем ему нужные стили и устанавливаем ширину табов

self.editor = CodeEditor()

self.editor.setStyleSheet("""QPlainTextEdit {

background-color: #222;

color: #A6B4C3;

font-family: Courier;

}"""

)

self.editor.setTabStopDistance(4 \* 8) # 4 - кол-во, 8 - ширина одного таба

# устанавилваем поле для редактирования кода центральным виджетом окна и меняем размер окна

self.setCentralWidget(self.editor)

self.resize(900, 500)

# МЕНЮ БАР --------------------------------------------------#

# создаем менюбар

menubar = self.menuBar()

#

# добавляем нужные действия и нужные атрибуты к ним для меню "File"

newFile = QAction('&New File', self)

newFile.setShortcut('Ctrl+N')

newFile.triggered.connect(self.newFile)

openFile = QAction('&Open', self)

openFile.setShortcut('Ctrl+O')

openFile.triggered.connect(self.showDialogOpenFile)

saveFile = QAction('&Save', self)

saveFile.setShortcut('Ctrl+S')

saveFile.triggered.connect(self.showDialogSaveFile)

# создаем меню "File" в менюбар и добавляем туда все нужные действия

fileMenu = menubar.addMenu('&File')

fileMenu.addAction(newFile)

fileMenu.addAction(openFile)

fileMenu.addAction(saveFile)

#

# добавляем нужные действия и нужные атрибуты к ним для меню "Edit"

undoAction = QAction("&Undo last action", self)

undoAction.setShortcut("Ctrl+Z")

undoAction.triggered.connect(self.Undo)

# создаем меню "Edit" в менюбар и добавляем туда все нужные действия

editMenu = menubar.addMenu('&Edit')

editMenu.addAction(undoAction)

# СТАТУС БАР ------------------------------------------------#

# создаем статусбар и назначаем ему вывод информации при изменении положения курсора

self.status = self.statusBar()

self.editor.cursorPositionChanged.connect(self.showCursorPosition)

# ДИАЛОГИ ---------------------------------------------------------------------------------------------------------#

# открытие файла

def showDialogOpenFile(self):

fileName, ok = QFileDialog.getOpenFileName(self, 'Open file', filter='\*.py')

if ok:

f = open(fileName, 'r')

with f:

data = f.read()

self.editor.setPlainText(data)

# сохранение файла

def showDialogSaveFile(self):

fileName, ok = QFileDialog.getSaveFileName(self, 'Save file', filter='\*.py')

if ok:

f = open(fileName, 'w')

with f:

data = self.editor.toPlainText()

f.write(data)

# создание нового файла

def newFile(self):

reply = QMessageBox.question(self, 'New File', "Are you want save this file?",

QMessageBox.Yes | QMessageBox.No, QMessageBox.Yes)

if reply == QMessageBox.Yes:

self.showDialogSaveFile()

self.editor.clear()

# ВНУТРЕННИЕ ФУНКЦИИ ----------------------------------------------------------------------------------------------#

# отображение в статусбаре текущее положение курсора (строка и линия)

def showCursorPosition(self):

line = self.editor.textCursor().blockNumber()

col = self.editor.textCursor().columnNumber()

linecol = 'Line ' + str(line+1) + ' | ' + 'Column ' + str(col+1)

self.status.showMessage(linecol)

# отмена последнего действия

def Undo(self):

self.editor.undo()

**Приложение 3**

#! usr/local/bin/python3

# -\*- coding: utf-8 -\*-

# README: поменяйте путь к Python, если на вашем компьютере он установлен в другую дирректорию

# main.py

import UI

import PythonHighlighter

import sys

from PyQt5.QtWidgets import QApplication

def main():

app = QApplication(sys.argv)

mainWindow = UI.MainWindow()

highlight = PythonHighlighter.PythonHighlighter(mainWindow.centralWidget().document())

mainWindow.show()

sys.exit(app.exec\_())

if \_\_name\_\_ == '\_\_main\_\_':

main()