Министерство образования Республики Беларусь

Учреждение образования

БелорусскиЙ государственный университет

информатики и радиоэлектроники

Факультет компьютерных систем и сетей

Кафедра информатики

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему: "Реализация среды визуального программирования (базовые функции: редактор текста, подключение компилятора, справка и т.п.)

Дисциплина: Операционные системы и среды (ОСиС)

Студент гр. 553502

В. В. Станько

Руководитель

ассистент кафедры информатики

В. В. Шнейдер

Минск 2018

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc513125742)

[1.ПОСТАНОВКА ЗАДАЧИ 4](#_Toc513125743)

[2.КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ 5](#_Toc513125744)

[2.1 Интегрированная среда разработки 5](#_Toc513125745)

[2.2 Среда визуальной разработки 6](#_Toc513125746)

[2.3 Используемые технологии 7](#_Toc513125747)

[2.3.1 Язык программирования Python 7](#_Toc513125748)

[2.3.2 PyQt 8](#_Toc513125749)

[2.3.3 SIP 9](#_Toc513125750)

[2.3.4 Особенности лицензирования PyQt 10](#_Toc513125751)

[2.3.5 Виртуальное окружение 10](#_Toc513125752)

[2.3.6 Virtualenv 11](#_Toc513125753)

[2.3.7 ini-конфигурации 11](#_Toc513125754)

[3. РЕАЛИЗАЦИЯ ЗАДАЧИ И ДЕМОНСТРАЦИЯ РАБОТЫ 13](#_Toc513125755)

[ЗАКЛЮЧЕНИЕ 22](#_Toc513125756)

[СПИСОК ИСПОЛЬЗОВАННЫХ РЕСУРСОВ 23](#_Toc513125757)

[ПРИЛОЖЕНИЕ 1. Код файла window.py 24](#_Toc513125758)

[ПРИЛОЖЕНИЕ 2. Код файла preferences.py 36](#_Toc513125759)

[ПРИЛОЖЕНИЕ 3. Код файла numberbar.py 47](#_Toc513125760)

[ПРИЛОЖЕНИЕ 4. Код файла interpreter\_dialog.py 51](#_Toc513125761)

[ПРИЛОЖЕНИЕ 5. Код файла highlighter.py 54](#_Toc513125762)

[ПРИЛОЖЕНИЕ 6. Код файла find.py 59](#_Toc513125763)

ВВЕДЕНИЕ

В современном мире всё чаще становится актуальным вопрос о разработке разнообразных приложений, а также внедрении их в жизнедеятельность человека.

Сегодня в каждой организации есть профессиональный компьютер с различными установленными приложениями, которые специалисту помогают в выполнении какой-либо работы. Также существуют развлекательные приложения, которые помогают работнику снять усталость и напряжение в перерыв между его профессиональной деятельностью. Приложения используются в производстве техники и оборудования. С помощью приложений происходит автоматизация производства. Часто приложения применяются в домашних устройствах: видеокамера, калькулятор, стиральная машина.

Для того, чтобы создать приложение, необходимо написать код программы. В наше время существует множество сред для визуального программирования, которые позволяют облегчить и ускорить написание кода для создания приложения.

Среда визуального программирования – неотъемлемая часть процесса разработки программного обеспечения. Она помогает ускорить процесс разработки программного обеспечения, поскольку включает в себя такие преимущества перед обычным блокнотом, как подсветка синтаксиса, поиск и замена слов в исходном тексте, подключение интерпретаторов и многое другое.

Таким образом, главной задачей данного курсового проекта является создание среды визуального программирования для языка Python.

1.ПОСТАНОВКА ЗАДАЧИ

Выполнить проект, в ходе которого будет реализована среда визуального программирования для языка Python, выполняющей следующие функции:

* Обеспечить редактирование текстовых документов с возможностью их последующего сохранения.
* Обеспечить возможность подсветки синтаксиса для выбранного языка (в частности, Python).
* Реализовать функцию, которая будет искать части текста в коде при помощи регулярных выражений.
* Обеспечить удобное управление пользователями.
* Определить возможность выбора файлов, а также папок с файлами для редактирования.
* Реализовать функцию по дополнению текста по введённой его части.
* Реализовать отслеживание состояния всех файлов, которые были открыты и сохранять их при завершении работы программы для того, чтобы не потерять данные.
* Реализовать функцию для изменения визуального оформления среды разработки.
* Реализовать возможность редактирования сразу нескольких файлов одновременно.
* Реализовать подключение интерпретатора, а также исполнения программы подключенным интерпретатором.

2.КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1 Интегрированная среда разработки

*Интегрированная среда разработки*, ИСP ([англ.](https://ru.wikipedia.org/wiki/Английский_язык) Integrated development environment — IDE), также единая среда разработки, ЕСР — комплекс программных средств, используемый [программистами](https://ru.wikipedia.org/wiki/Программист) для разработки [программного обеспечения](https://ru.wikipedia.org/wiki/Программное_обеспечение) (ПО).

Среда разработки включает в себя:

* текстовый редактор;
* [компилятор](https://ru.wikipedia.org/wiki/Компилятор) и/или интерпретатор;
* средства автоматизации сборки;
* [отладчик](https://ru.wikipedia.org/wiki/Отладчик).

Иногда содержит также средства для интеграции с [системами управления версиями](https://ru.wikipedia.org/wiki/Система_управления_версиями) и разнообразные инструменты для упрощения конструирования [графического интерфейса пользователя](https://ru.wikipedia.org/wiki/Графический_интерфейс_пользователя). Многие современные среды разработки также включают браузер классов, инспектор объектов и диаграмму [иерархии классов](https://ru.wikipedia.org/wiki/Иерархия_классов) — для использования при [объектно-ориентированной разработке](https://ru.wikipedia.org/wiki/Объектно-ориентированное_проектирование) ПО. Интегрированные среды разработки обычно предназначены для нескольких [языков программирования](https://ru.wikipedia.org/wiki/Язык_программирования) — такие как [IntelliJ IDEA](https://ru.wikipedia.org/wiki/IntelliJ_IDEA), [NetBeans](https://ru.wikipedia.org/wiki/NetBeans), [Eclipse](https://ru.wikipedia.org/wiki/Eclipse_(среда_разработки)), [Qt Creator](https://ru.wikipedia.org/wiki/Qt_Creator), [Geany](https://ru.wikipedia.org/wiki/Geany), [Embarcadero RAD Studio](https://ru.wikipedia.org/wiki/Embarcadero_RAD_Studio), [Code::Blocks](https://ru.wikipedia.org/wiki/Code::Blocks), [Xcode](https://ru.wikipedia.org/wiki/Xcode) или [Microsoft Visual Studio](https://ru.wikipedia.org/wiki/Microsoft_Visual_Studio), но есть и IDE для одного определённого языка программирования — как, например, [Visual Basic](https://ru.wikipedia.org/wiki/Visual_Basic), [Delphi](https://ru.wikipedia.org/wiki/Delphi_(среда_разработки)), [Dev-C++](https://ru.wikipedia.org/wiki/Dev-C%2B%2B).

Использование ИСР для разработки программного обеспечения является прямой противоположностью способу, в котором используются несвязанные инструменты, такие как текстовый редактор, [компилятор](https://ru.wikipedia.org/wiki/Компилятор), и т. п. Интегрированные среды разработки были созданы для того, чтобы максимизировать производительность программиста благодаря тесно связанным компонентам с простыми пользовательскими интерфейсами. Это позволяет разработчику сделать меньше действий для переключения различных режимов, в отличие от дискретных программ разработки. Однако так как ИСР является сложным программным комплексом, то среда разработки сможет качественно ускорить процесс разработки [ПО](https://ru.wikipedia.org/wiki/Программное_обеспечение) лишь после специального обучения. Для уменьшения барьера вхождения многие достаточно интерактивны, а для облегчения перехода с одной на другую интерфейс у одного производителя максимально близок, вплоть до использования одной ИСР.

ИСР обычно представляет собой единственную программу, в которой проводится вся разработка. Она, как правило, содержит много функций для создания, изменения, компилирования, развертывания и отладки программного обеспечения. Цель интегрированной среды заключается в том, чтобы объединить различные утилиты в одном модуле, который позволит абстрагироваться от выполнения вспомогательных задач, тем самым позволяя программисту сосредоточиться на решении собственно алгоритмической задачи и избежать потерь времени при выполнении типичных технических действий (например, вызове компилятора). Таким образом, повышается производительность труда разработчика. Также считается, что тесная интеграция задач разработки может далее повысить производительность за счёт возможности введения дополнительных функций на промежуточных этапах работы. Например, ИСР позволяет проанализировать код и тем самым обеспечить мгновенную обратную связь и уведомить о синтаксических ошибках.

Большинство современных ИСР являются графическими. Но первые ИСР использовались ещё до того, как стали широко применяться операционные системы с графическим интерфейсом — они были основаны на текстовом интерфейсе с использованием функциональных и горячих клавиш для вызова различных функций (например, [Turbo Pascal](https://ru.wikipedia.org/wiki/Turbo_Pascal), созданный фирмой [Borland](https://ru.wikipedia.org/wiki/Borland)).

Частный случай интегрированной среды разработки — [среды визуальной разработки](https://ru.wikipedia.org/w/index.php?title=Среда_визуальной_разработки&action=edit&redlink=1), которые включают в себя возможность [наглядного редактирования](https://ru.wikipedia.org/wiki/Визуальное_программирование) интерфейса программы.

2.2 Среда визуальной разработки

Среда визуальной разработки – это среда разработки программного обеспечения, в которой наиболее распространенные блоки программного кода представлены в виде графических объектов. Применяются в основном для создания прикладных программ и разработки графического интерфейса пользователя (GUI).

2.2.1 Преимущества и недостатки сред

Преимущества:

* быстрота разработки;
* лёгкость освоения;
* стандартизация внешнего вида программ.

Недостатки:

* как правило, привязка к конкретной среде разработки непосредственно связана с проблематичностью перехода на другую среду разработки;
* затруднённое использование нестандартных компонентов;
* наличие недокументированных особенностей компонент.

Следует учитывать, что некоторые визуальные среды разработки имеют собственный формат хранения проекта и при переходе на другую среду может возникнуть непереносимость свойств проекта и некоторых частей проекта, таких, как собственные библиотеки используемой среды разработки.

Также следует учитывать некоторые изменения, которые могут вносится в язык программирования. Конкретный пример - среда разработки Delphi (за основу взят Pascal, но по сути это уже новый язык программирования). Среду разработки, как и язык программирования следует выбирать на этапе проектирования ПО. Правильно спроектированное ПО должно учитывать развитие и внедрение новых технологий, поэтому перенос разработки такого ПО в другую среду разработки не должен представлять трудностей.

2.3 Используемые технологии

2.3.1 Язык программирования Python

*Python* — [высокоуровневый язык программирования](https://ru.wikipedia.org/wiki/Высокоуровневый_язык_программирования) общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. [Синтаксис](https://ru.wikipedia.org/wiki/Синтаксис_(программирование)) ядра Python минималистичен. В то же время [стандартная библиотека](https://ru.wikipedia.org/wiki/Стандартная_библиотека_Python) включает большой объём полезных функций[1].

Python поддерживает несколько парадигм программирования, в том числе структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное. Основные архитектурные черты — динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений и удобные высокоуровневые структуры данных. Код в Python организовывается в функции и классы, которые могут объединяться в модули (они в свою очередь могут быть объединены в пакеты).

Эталонной реализацией Python является интерпретатор CPython, поддерживающий большинство активно используемых платформ. Он распространяется под свободной лицензией Python Software Foundation License, позволяющей использовать его без ограничений в любых приложениях, включая проприетарные. Есть реализации интерпретаторов для JVM (с возможностью компиляции), MSIL (с возможностью компиляции), LLVM и других. Проект PyPy предлагает реализацию Python с использованием JIT-компиляции, которая значительно увеличивает скорость выполнения Python-программ.

Python — активно развивающийся [язык программирования](https://ru.wikipedia.org/wiki/Язык_программирования), новые версии (с добавлением/изменением языковых свойств) выходят примерно раз в два с половиной года. Вследствие этого и некоторых других причин на Python отсутствуют [стандарт ANSI](https://ru.wikipedia.org/wiki/Американский_национальный_институт_стандартов), [ISO](https://ru.wikipedia.org/wiki/ISO) или другие официальные стандарты, их роль выполняет [CPython](https://ru.wikipedia.org/wiki/CPython).

Разработчики языка Python придерживаются определённой философии программирования, называемой «The Zen of Python» («[Дзен](https://ru.wikipedia.org/wiki/Дзен) Питона», или «Дзен Пайтона»). Её текст выдаётся [интерпретатором](https://ru.wikipedia.org/wiki/Интерпретатор) Python по команде import this (работает один раз за сессию). Автором этой философии считается [Тим Петерс](https://ru.wikipedia.org/w/index.php?title=Тим_Петерс&action=edit&redlink=1) (Tim Peters).

Текст философии:

* Красивое лучше, чем уродливое.
* Явное лучше, чем неявное.
* Простое лучше, чем сложное.
* Сложное лучше, чем запутанное.
* Плоское лучше, чем вложенное.
* Разреженное лучше, чем плотное.
* Читаемость имеет значение.
* Особые случаи не настолько особые, чтобы нарушать правила.
* При этом практичность важнее безупречности.
* Ошибки никогда не должны замалчиваться.
* Если не замалчиваются явно.
* Встретив двусмысленность, отбрось искушение угадать.
* Должен существовать один — и, желательно, только один — очевидный способ сделать это.
* Хотя он поначалу может быть и не очевиден, если вы не голландец.
* Сейчас лучше, чем никогда.
* Хотя никогда зачастую лучше, чем прямо сейчас.
* Если реализацию сложно объяснить — идея плоха.
* Если реализацию легко объяснить — идея, возможно, хороша.
* Пространства имён — отличная вещь! Давайте будем делать их больше!

2.3.2 PyQt

PyQt — набор «[привязок](https://ru.wikipedia.org/w/index.php?title=Привязка_(программирование)&action=edit&redlink=1)» [графического](https://ru.wikipedia.org/wiki/Графический_интерфейс_пользователя) [фреймворка](https://ru.wikipedia.org/wiki/Фреймворк) [Qt](https://ru.wikipedia.org/wiki/Qt) для [языка программирования](https://ru.wikipedia.org/wiki/Язык_программирования) [Python](https://ru.wikipedia.org/wiki/Python), выполненный в виде [расширения](https://ru.wikipedia.org/wiki/Плагин) Python[10]. PyQt разработан [британской](https://ru.wikipedia.org/wiki/Великобритания) компанией Riverbank Computing. PyQt работает на всех платформах, поддерживаемых Qt: [Linux](https://ru.wikipedia.org/wiki/Linux) и другие [UNIX-подобные ОС](https://ru.wikipedia.org/wiki/Unix-подобная_операционная_система), [Mac OS X](https://ru.wikipedia.org/wiki/Mac_OS_X) и [Windows](https://ru.wikipedia.org/wiki/Windows). Существует 2 версии: PyQt5, поддерживающий Qt 5, и PyQt4, поддерживающий Qt 4. PyQt распространяется под лицензиями [GPL](https://ru.wikipedia.org/wiki/GPL) (2 и 3 версии) и [коммерческой](https://ru.wikipedia.org/wiki/Проприетарное_программное_обеспечение).

PyQt практически полностью реализует возможности Qt. А это более 600 классов, более 6000 функций и методов, включая:

* существующий набор [виджетов](https://ru.wikipedia.org/wiki/Виджет) графического интерфейса;
* стили виджетов;
* доступ к [базам данных](https://ru.wikipedia.org/wiki/База_данных) с помощью [SQL](https://ru.wikipedia.org/wiki/SQL) ([ODBC](https://ru.wikipedia.org/wiki/ODBC), [MySQL](https://ru.wikipedia.org/wiki/MySQL), [PostgreSQL](https://ru.wikipedia.org/wiki/PostgreSQL), [Oracle](https://ru.wikipedia.org/wiki/Oracle_(СУБД)));
* QScintilla, основанный на [Scintilla](https://ru.wikipedia.org/wiki/Scintilla) виджет текстового редактора;
* поддержку [интернационализации](https://ru.wikipedia.org/wiki/Интернационализация) (i18n);
* [парсер](https://ru.wikipedia.org/wiki/Парсер) XML (Extensible Markup Language);
* поддержку [SVG](https://ru.wikipedia.org/wiki/SVG);
* интеграцию с [WebKit](https://ru.wikipedia.org/wiki/WebKit), движком рендеринга HTML;
* поддержку воспроизведения видео и аудио.

PyQt также включает в себя [Qt Designer](https://ru.wikipedia.org/wiki/Qt_Designer) ([Qt Creator](https://ru.wikipedia.org/wiki/Qt_Creator)) — дизайнер графического интерфейса пользователя. Программа pyuic генерирует Python код из файлов, созданных в Qt Designer. Это делает PyQt очень полезным инструментом для быстрого прототипирования. Кроме того, можно добавлять новые графические элементы управления, написанные на Python, в Qt Designer.

Раньше PyQt поставлялся вместе со [средой разработки](https://ru.wikipedia.org/wiki/Интегрированная_среда_разработки) [Eric](https://ru.wikipedia.org/wiki/Eric_Python_IDE), написанной на PyQt. Eric имеет встроенный отладчик и может быть использована для создания консольных программ. Теперь она доступна в качестве отдельного проекта.

Обзор основных модулей[3]:

* *QtCore* — основные не графические классы: система сигналов и слотов, независимые от платформ абстракции для [Unicode](https://ru.wikipedia.org/wiki/Unicode), потоков, разделяемой памяти, [регулярных выражений](https://ru.wikipedia.org/wiki/Регулярные_выражения) и т. д.
* *QtGui* — компоненты [графического интерфейса](https://ru.wikipedia.org/wiki/Графический_интерфейс_пользователя) (элементы управления), основанные на визуальном представлении.
* *QtNetwork* — классы для сетевого программирования. Например, клиентов и серверов через [UDP](https://ru.wikipedia.org/wiki/UDP) и [TCP](https://ru.wikipedia.org/wiki/TCP).
* *QtOpenGL* — классы, позволяющие использовать [OpenGL](https://ru.wikipedia.org/wiki/OpenGL) и 3D-графику в приложениях PyQt.
* *QtScript* — классы, позволяющие использовать встроенный в Qt интерпретатор [JavaScript](https://ru.wikipedia.org/wiki/JavaScript) для управления приложением.
* *QtSql* — классы для интеграции с базами данных с помощью SQL.
* *QtSvg* — классы для отображения векторной графики в формате [SVG](https://ru.wikipedia.org/wiki/SVG).
* *QtXml* — классы, реализующие обработку XML.
* *uic* — реализация обработки XML-файлов, созданных в Qt Designer, для генерации из них Python-кода графического интерфейса.

2.3.3 SIP

[SIP](https://ru.wikipedia.org/w/index.php?title=SIP_(инструмент_PyQt)&action=edit&redlink=1) — это инструмент для автоматической генерации Python-привязок к C и C++ библиотекам. Первоначально SIP был разработан в 1998 году для PyQt — Python-привязки для инструментария виджетов Qt, но подходит и для создания привязок к любым другим C или C++ библиотекам[]](https://ru.wikipedia.org/wiki/PyQt#cite_note-sip.introduction-8).

SIP берёт набор инструкций (.sip файлы), описывающих [API](https://ru.wikipedia.org/wiki/API) и генерирует требуемый C++ код. Потом он компилируется и на выходе получается модуль Python. Файлы .sip — файлы заголовков класса, у которых кое-что убрано (потому что SIP не содержит полноценный C++-парсер) и кое-что добавлено (поскольку C++ не несёт достаточной информации о работе API).

Для PyQt используется внутренний инструмент (написанный на PyQt), называемый metasip. Это своего рода IDE для SIP. Он использует [GCC](https://ru.wikipedia.org/wiki/GCC)-XML для разбора заголовочных файлов последней версии и сохраняет релевантные данные в XML(Extensible Markup Language), в metasip-проект. metasip далее делает некое подобие diff с предыдущей версией API и отмечает все изменения, которые необходимо просмотреть. Затем список изменений генерируется при помощи GUI и автоматически заносится в [TODO](https://ru.wikipedia.org/wiki/TODO). Создание .sip файлов — просто нажатие на кнопку.

Что касается работы сгенерированного кода, она не сильно отличается от работы любого другого генератора привязок. У Python очень хороший API модулей — это одна из причин большого количества Python-привязок для инструментов сторонних разработчиков. Для каждого C++-класса SIP генерирует C-код, создающий соответствующий Python-класс[.](https://ru.wikipedia.org/wiki/PyQt#cite_note-интервью-9)

2.3.4 Особенности лицензирования PyQt

Двойная [лицензия](https://ru.wikipedia.org/wiki/Лицензия_на_программное_обеспечение) PyQt продиктована довольно сложными отношениями в сфере применения этой надстройки (привязки) над графической библиотекой Qt. В соответствии с лицензией [GPL](https://ru.wikipedia.org/wiki/GPL), возможно использование библиотеки для создания программ с [открытым исходным кодом](https://ru.wikipedia.org/wiki/Открытый_исходный_код). Если нет возможности распространять код в соответствии с условиями GPL, то можно приобрести коммерческую лицензию[.](https://ru.wikipedia.org/wiki/PyQt#cite_note-осн.лиц-10)

Со стороны [Nokia](https://ru.wikipedia.org/wiki/Nokia) были попытки договориться с Riverbank Computing на предмет смены лицензии, но они не увенчались успехом. В результате возник проект под названием [PySide](https://ru.wikipedia.org/wiki/PySide) — аналог PyQt, в том числе сохраняющий совместимость с последним на уровне API, но выпущенный под лицензией [LGPL](https://ru.wikipedia.org/wiki/LGPL), чтобы дать возможность разработчикам и коммерческих проектов бесплатно использовать Python в связке с Qt.

Однако Riverbank не исключает возможности лицензирования PyQt под LGPL в будущем.

2.3.5 Виртуальное окружение

Виртуальное окружение для языка Python — очень удобный инструмент при одновременной работе с несколькими проектами. При разработке устанавливаются различные библиотеки, да и версия самого Python может отличаться[6].

Использование виртуального окружения позволяет абстрагироваться от библиотек, используемых в системе. Для начала создается виртуальное окружение, потом оно активируется, и все устанавливаемые модули Питона будут установлены в данном виртуальном окружении. Чтобы работать в другом проекте с другими версиями библиотек, достаточно просто переключить виртуальное окружение. Для работы с виртуальным окружением понадобятся два инструмента: virtualenv и virtualenvwrapper.

2.3.6 Virtualenv

Virtualenv — это инструмент, позволяющий создавать виртуальные окружения с пакетами [9]. Разные «песочницы» (например, напоминает sandbox’ы в [Cabal](https://eax.me/cabal/)) имеют разный набор пакетов разных версий. Работая над конкретным проектом, запросто можно переключаться на подходящую песочницу, и проблема уходит. Несложно заметить, что в отличие от того, как это сделано в некоторых других языках программирования, одну и ту же песочницу можно смело использовать сразу в нескольких проектах. Плюс к этому уходит проблема засорения системы ненужными пакетами, так как песочницы можно легко создавать и удалять.

2.3.7 ini-конфигурации

ini-файл — это файл конфигурации, который содержит данные настроек для Microsoft Windows, Windows NT и некоторых приложений.

Появились с самых первых версий Windows. В версии Windows 1.01 это был только файл WIN.INI. В Windows 3.0 добавился файл SYSTEM.INI. А затем их количество начало расти быстро и бесконтрольно.

Не существует подробной официальной спецификации формата. Начиная с Windows 95, INI файлы считаются устаревшими и в качестве замены им Microsoft предлагает использовать системный реестр (Registry). Тем не менее INI файлы продолжают использоваться как приложениями других производителей, так и компонентами ОС от Microsoft. Например, файл boot.ini используется в Windows NT4/2000/XP при загрузке для выбора из нескольких ОС.

Хотя INI файлы и приобрели популярность в Windows, они могут использоваться в любой ОС. Несложная структура этого формата позволяет легко обрабатывать их программно и имеет достаточно понятный вид для чтения и изменения человеком. Существуют текстовые редакторы (например, SciTE или Notepad++), которые используют подсветку синтаксиса и фолдинг для этого формата, что облегчает редактирование INI файлов вручную.

Использование секций для разделения параметров способствует систематизации данных, однако для хранения настроек большого объёма с более сложной иерархической структурой лучше подходит формат XML  или JSON (JavaScript Object Notation).

3. РЕАЛИЗАЦИЯ ЗАДАЧИ И ДЕМОНСТРАЦИЯ РАБОТЫ

В данной главе произведем демонстрацию основных возможностей программы, а именно:

* Возможность обзора и изменения конфигураций приложения;
* Открытие и редактирование файлов;
* Отображение автодополнения и подсветки кода;
* Возможность поиска и замены по шаблону в тексте.

При запуске приложения появится окно с областью для отображения файлов в папке, областью для отображения текстовых файлов, панелью меню(Рис. 3.1).

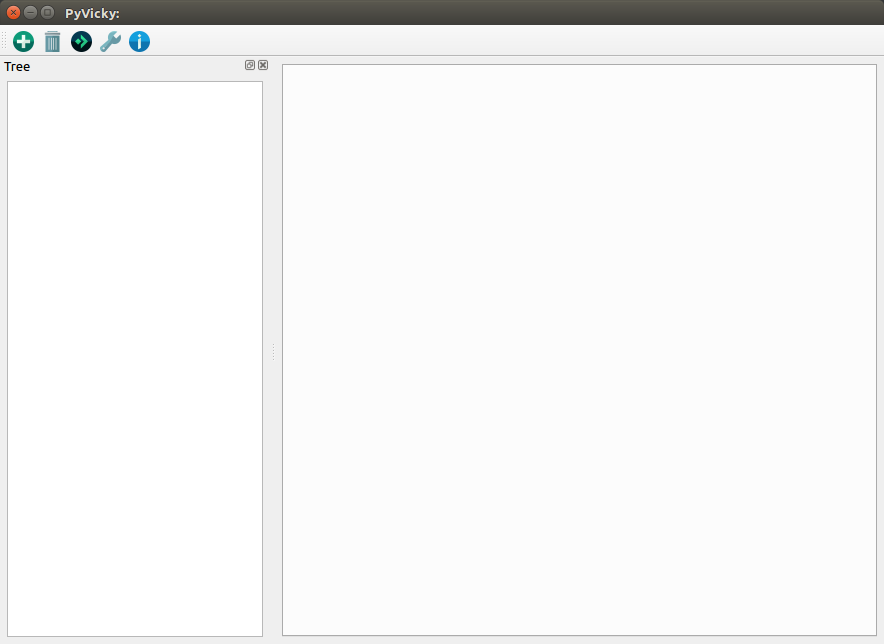


Рисунок 3.1 Интерфейс пользователя при старте программы

При нажатии комбинации клавиш Ctrl + N, или кнопки New на панели меню, или File→New в панели контекстного меню можно увидеть появившуюся панель для написания текста (см. Рис. 3.2).



Рисунок 3.2 Создание нового текстового документа

Если нажать в контекстном меню File → Open file или нажать комбинацию клавиш Сtrl+O, то можно открыть уже существующие файлы для редактирования[7] (Рис. 3.3).

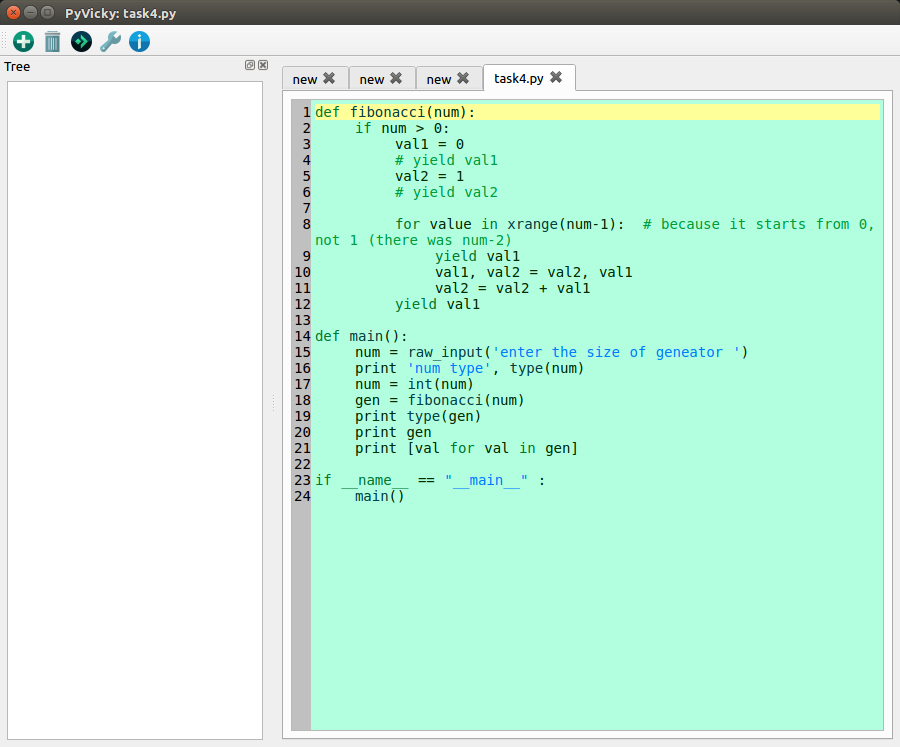


Рисунок 3.3 Интерфейс пользователя при открытии нескольких вкладок с текстовыми файлами одновременно

Для того, чтобы настроить конфигурации интерпретатора, необходимо зайти в диалоговое меню настройки при помощи комбинации клавиш Ctrl+I или нажать в контекстном меню Tools → Interpreter Settings или нажать значок гаечного ключа на панели меню. В данном меню можно настроить путь к интерпретатору, путь к файлу для интерпретации и аргументы командной строки.

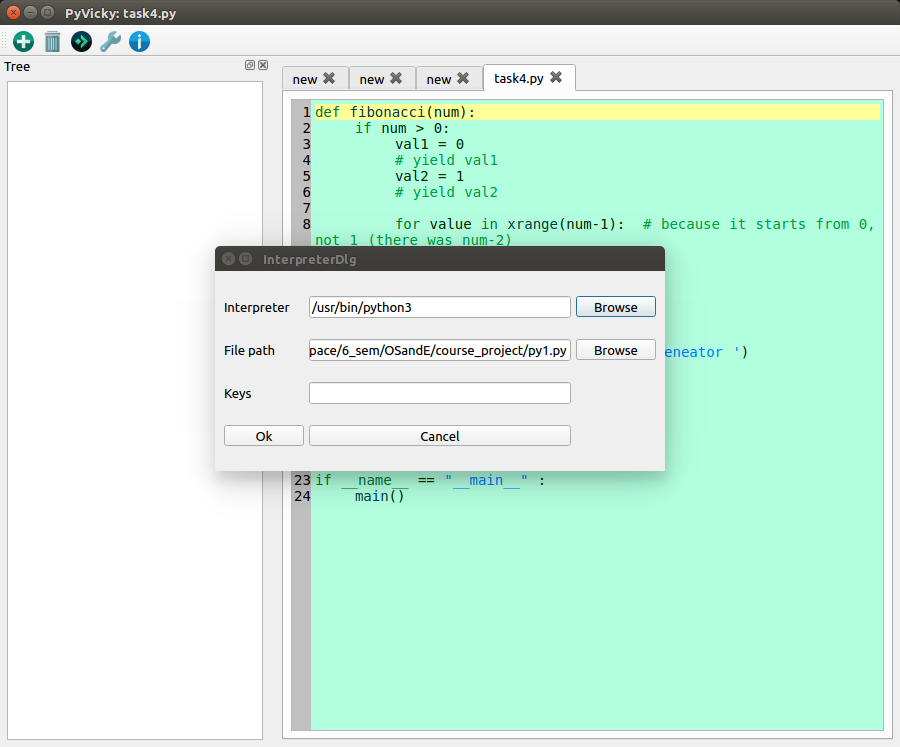


Рисунок 3.4 Диалоговое окно конфигураций интерпретатора

Для того, чтобы проводить определенные манипуляции с текстом, такие, как поиск по слову, поиск по регулярному выражению и замена элементов в тексте, можно вызвать диалоговое окно поиска, нажав Ctrl+F или выбрав пункт в контекстном меню Search → Search (Рис. 3.5).

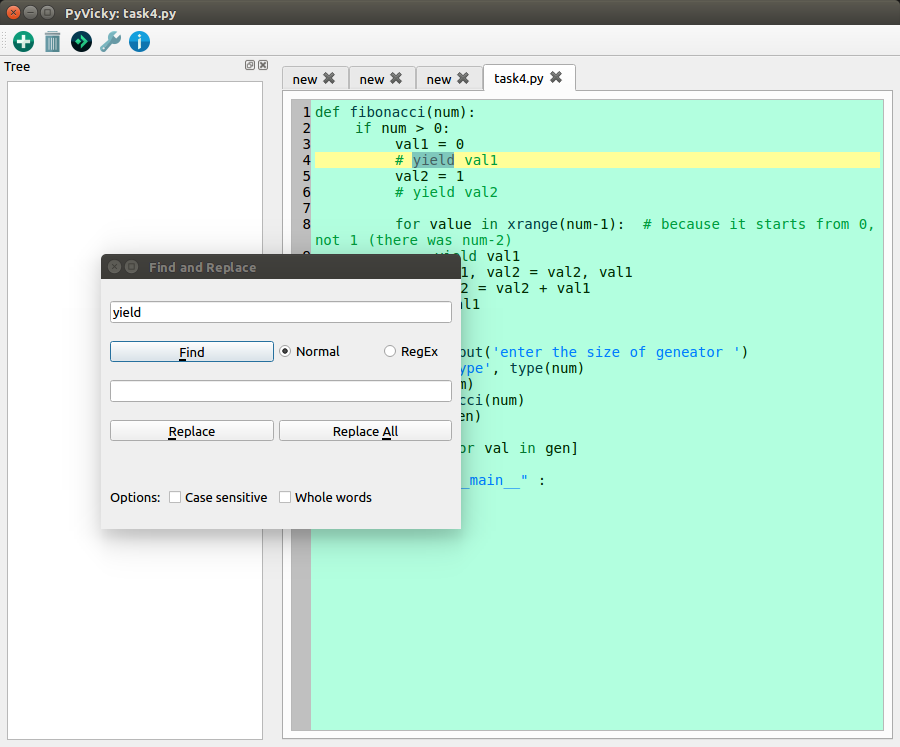


Рисунок 3.5 Диалоговое окно поиска и замены в тексте

Если нажать в контекстном меню File → Open directory или нажать комбинацию клавиш Сtrl+Shift+O, то можно открыть уже существующие файлы для редактирования, в области слева отобразятся файлы и папки, которые находятся в открытой папке (Рис. 3.6).

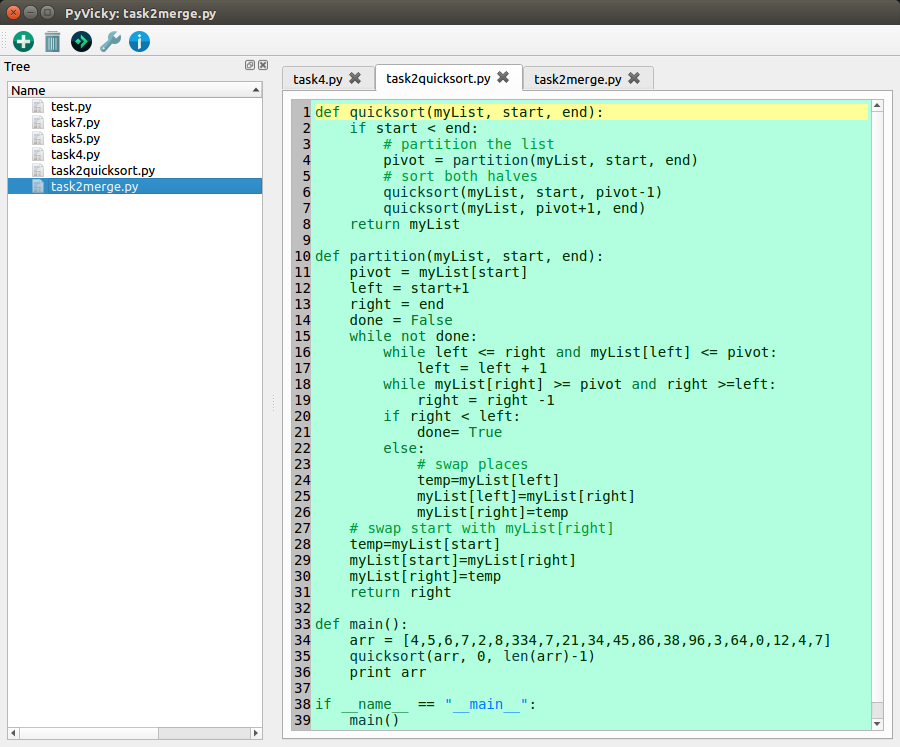


Рисунок 3.6 Интерфейс пользователя при открытии папки, в которой лежат текстовые файлы

Для того, чтоб изменить внешний вид и основную тему текстового редактора, сменить тип шрифта и его размер, настроить подсветку синтаксиса[8], необходимо зайти в диалоговое окно настроек при помощи комбинации клавиш Ctrl+P или в контекстном меню View → Preferences (Рис. 3.6).

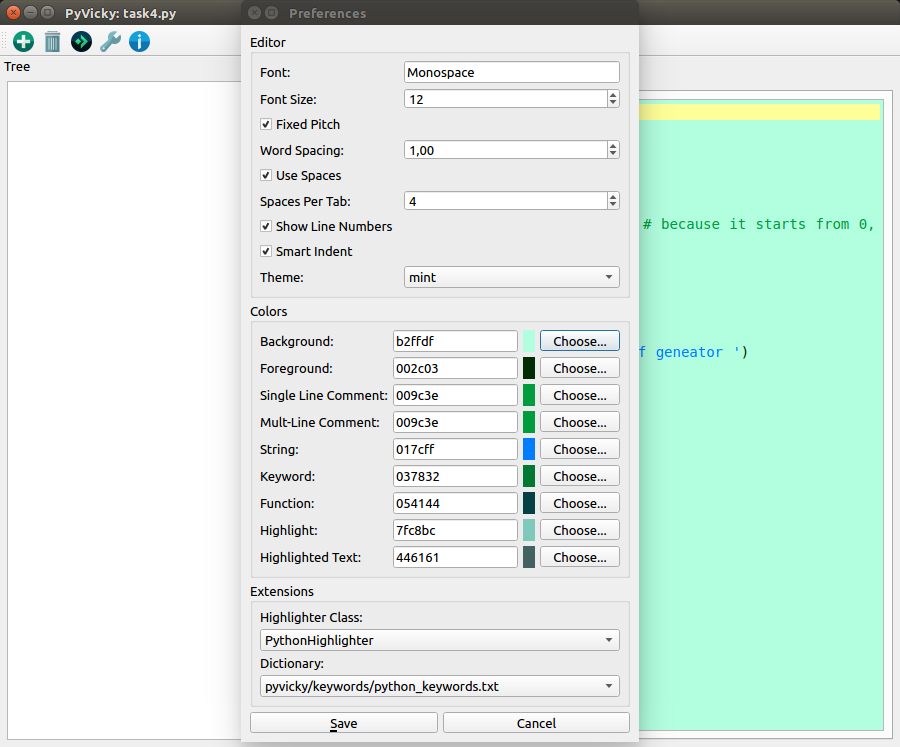


Рисунок 3.7 Диалоговое окно выбора внешнего вида текстового редактора и подсветки кода

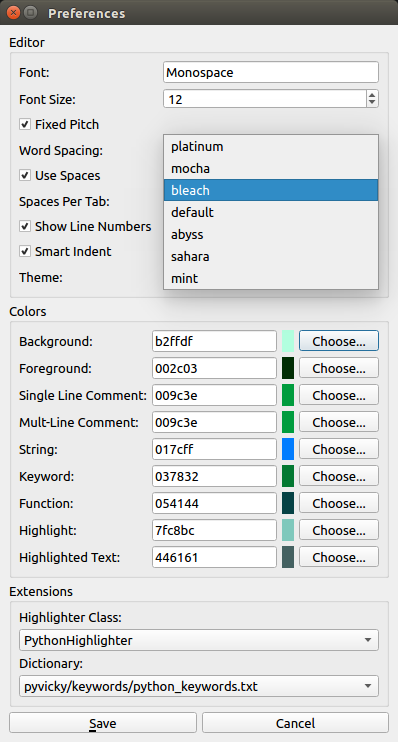


Рисунок 3.8 Выбор цветовой темы редактора

Во время написания кода с целью увеличения быстродействия кода был разработан модуль, который автоматически дополняет неполные ключевые слова. Автоматическое дополнение вызывается самостоятельно при написании текста, но можно вызвать его вручную при помощи комбинации Ctrl+E (Рис. 3.9).

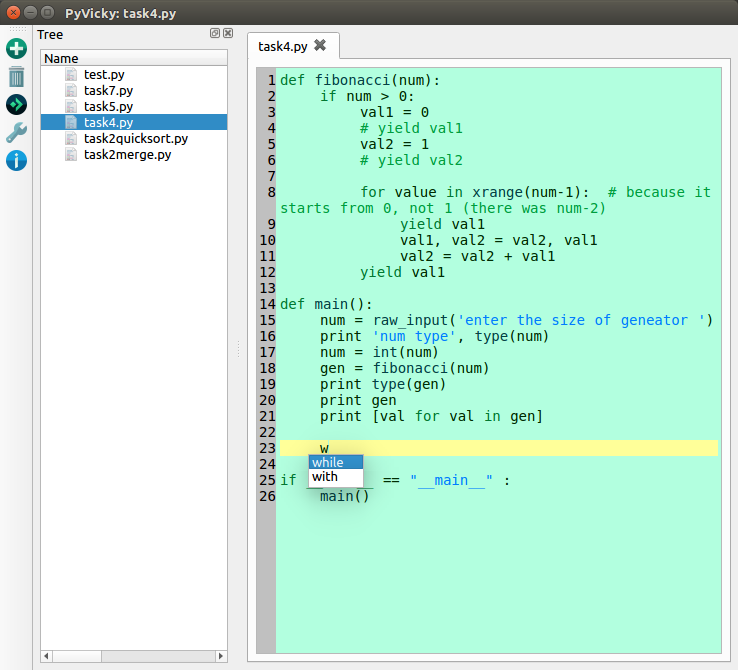


Рисунок 3.9 Автоматическое дополнение кода

ЗАКЛЮЧЕНИЕ

В ходе работы над курсовым проектом был разработан программный продукт для написания, сохранения, редактирования кода на языке Python. Созданная среда визуального программирования также позволяет интерпретировать написанный код, выполнять поиск и замену в коде, использовать автоматическое дополнение кода для улучшения качества и скорости написания кода.

В результате написания данного проекта были получены и закреплены навыки работы с библиотекой PyQt 5, виртуальным окружением virtualenv, системой контроля версий, а также изучены методы работы с конфигурационными ini-файлами.

Созданный проект был разделен на основные логические модули, которые связаны между собой инфраструктурой для более удобной разработки и дальнейшей поддержки.

Программное средство, которое было создано в результате выполнения данной курсовой работы, - это проектный программный продукт, готовый к использованию на персональных компьютерах различных типов.

Программный продукт, разработанный в ходе работы над курсовым проектом, имеет ряд преимуществ над своими аналогами, поскольку он имеет открытый исходный код, является бесплатным, имеет большой размер рабочей области, способен работать на многих операционных системах, таких как Windows, Linux, Mac OS.

В качестве улучшений видятся такие направления, как добавление новых языков и подсветок их синтаксиса в приложении, добавление утилит для возможности отладки приложения, добавления синтаксического, лексического и семантического анализов написанного кода, общее расширение интерфейса и функциональности.

СПИСОК ИСПОЛЬЗОВАННЫХ РЕСУРСОВ

1. Википедия [Интернет ресурс]. - Электронные данные. - Режим доступа: http://ru. wikipedia.org/wiki/. Дата доступа – 13.04.2018
2. ConfigParser [Интернет ресурс]. - Электронные данные. - Режим доступа: https://docs.python.org/2/library/configparser.html. Дата доступа – 13.04.2018
3. PyQt5 modules [Интернет ресурс]. - Электронные данные. - Режим доступа: http://pyqt.sourceforge.net/Docs/PyQt5/modules.html. Дата доступа – 13.04.2018
4. Создание Python пакетов [Интернет ресурс]. - Электронные данные. - Режим доступа: http://klen.github.io/create-python-packages.html. Дата доступа – 13.04.2018
5. PyQt5 [Интернет ресурс]. - Электронные данные. - Режим доступа: https://pythonworld.ru/gui/pyqt5-firstprograms.html. Дата доступа – 13.04.2018
6. Setting up virtual environment [Интернет ресурс]. - Электронные данные. - Режим доступа: https://www.digitalocean.com/community/tutorials/how-to-install-python-3-and-set-up-a-programming-environment-on-an-ubuntu-16-04-server. Дата доступа – 13.04.2018
7. PyQt Tabs [Интернет ресурс]. - Электронные данные. - Режим доступа: https://www.tutorialspoint.com/pyqt/pyqt\_qtabwidget.htm Дата доступа – 13.04.2018
8. PyQt Highlighting [Интернет ресурс]. - Электронные данные. - Режим доступа: https://wiki.python.org/moin/PyQt/Python%20syntax%20highlighting. Дата доступа – 13.04.2018
9. Installing using pip and virtualenv [Интернет ресурс]. - Электронные данные. - Режим доступа: https://packaging.python.org/guides/installing-using-pip-and-virtualenv/ Дата доступа – 13.04.2018
10. PyQt [Интернет ресурс]. - Электронные данные. - Режим доступа: [https://ru.wikipedia.org/wiki/PyQt. Дата доступа – 13.04.2018](https://ru.wikipedia.org/wiki/PyQt.%20Дата%20доступа%20–%2013.04.2018)

**ПРИЛОЖЕНИЕ 1. Код файла window.py**

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

import configparser

import sys

import os

from PyQt5.QtCore import Qt

from PyQt5.QtWidgets import QApplication, QWidget, QAction, QMainWindow, QInputDialog, QLineEdit, QFileDialog, \

QTextEdit, QMessageBox, QVBoxLayout, QTabWidget, QFileSystemModel, QTreeView, QDockWidget

from PyQt5.QtGui import QIcon, QFont, QSyntaxHighlighter, QTextCharFormat

from pyvicky.highlighter import PythonHighlighter

from pyvicky.preferences import PreferencesDlg

from pyvicky.numberbar import QCodeEditor

from pyvicky.find import Find

from pyvicky.interpreter\_dialog import InterpreterDlg

import traceback

import subprocess

import logging

import re

import time

logger = logging.getLogger('window logger')

ini\_pattern = re.compile('(.)\*\.ini')

class Window(QMainWindow):

"""

Main window class

"""

def \_\_init\_\_(self, x, y, filename='unknown'):

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

self.resize(x, y)

self.move(300, 300)

self.setWindowTitle('PyVicky: ' + filename)

self.setWindowIcon(QIcon('pyvicky/staticfiles/pencil.png'))

logger.info('Window created')

self.toolBar = {}

self.highlighter = {}

self.currentFileName = 'Untitled'

self.currentFilePath = os.getcwd()

self.firstSave = True

self.findDlg = None

self.add\_menu\_bar()

self.add\_tool\_bar()

self.docked\_items = QDockWidget("Tree", self)

self.treeWidget = DirectoriesTreeView(self)

self.treeWidget.tree.clicked.connect(self.open\_file\_from\_tree)

self.docked\_items.setWidget(self.treeWidget)

self.docked\_items.setFloating(False)

# self.text = QTextEdit(self)

# self.text = QCodeEditor(self)

# self.setCentralWidget(self.text)

self.tabWidget = TabView(self)

self.setCentralWidget(self.tabWidget)

self.tabWidget.tabWidget.tabCloseRequested.connect(self.close\_tab)

self.addDockWidget(Qt.LeftDockWidgetArea, self.docked\_items)

# self.tabWidget.add\_tab(self, "new")

self.font = QFont()

self.settings = configparser.ConfigParser()

self.config = configparser.ConfigParser()

self.setup\_editor()

self.show()

def add\_menu\_bar(self):

main\_menu = self.menuBar()

file\_menu = main\_menu.addMenu('File')

edit\_menu = main\_menu.addMenu('Edit')

view\_menu = main\_menu.addMenu('View')

search\_menu = main\_menu.addMenu('Search')

tools\_menu = main\_menu.addMenu('Tools')

help\_menu = main\_menu.addMenu('Help')

exit\_button = QAction('Exit', self)

exit\_button.setShortcut('Ctrl+Q')

exit\_button.setStatusTip('Exit application')

exit\_button.triggered.connect(self.close) # replace self.close to normal method(which is need)

open\_button = QAction('Open file', self)

open\_button.setShortcut('Ctrl+O')

open\_button.setStatusTip('Open file')

open\_button.triggered.connect(self.open\_file) # without brackets, else it will execute immediately

new\_button = QAction('New file', self)

new\_button.setShortcut('Ctrl+N')

new\_button.setStatusTip('New file')

new\_button.triggered.connect(self.new\_file)

save\_button = QAction('Save file', self)

save\_button.setShortcut('Ctrl+S')

save\_button.setStatusTip('Save file')

save\_button.triggered.connect(self.save\_file)

open\_dir\_button = QAction('Open directory', self)

open\_dir\_button.setShortcut('Ctrl+Shift+O')

open\_dir\_button.setStatusTip('Open directory')

open\_dir\_button.triggered.connect(self.open\_directory)

file\_menu.addAction(exit\_button)

file\_menu.addAction(open\_button)

file\_menu.addAction(new\_button)

file\_menu.addAction(save\_button)

file\_menu.addAction(open\_dir\_button)

copy\_text = QAction('Copy', self)

paste\_text = QAction('Paste', self)

clear\_text = QAction('Clear', self)

copy\_text.setShortcut('Ctrl+C')

copy\_text.triggered.connect(self.copy\_func)

paste\_text.setShortcut('Ctrl+V')

paste\_text.triggered.connect(self.paste\_func)

# Ctrl+A works by itself

clear\_text.setShortcut('Ctrl+Y')

clear\_text.triggered.connect(self.clear\_text)

edit\_menu.addAction(copy\_text)

edit\_menu.addAction(paste\_text)

edit\_menu.addAction(clear\_text)

edit\_highlight = QAction('Preferences', self)

edit\_highlight.setShortcut('Ctrl+P')

edit\_highlight.triggered.connect(self.change\_preferences)

view\_menu.addAction(edit\_highlight)

help\_edit = QAction('Help', self)

help\_edit.setShortcut('Ctrl+H')

help\_edit.triggered.connect(self.about)

help\_menu.addAction(help\_edit)

help\_menu.addAction("About Qt", QApplication.instance().aboutQt)

search\_edit = QAction('Search', self)

search\_edit.setShortcut('Ctrl+F')

search\_edit.triggered.connect(self.search\_config)

search\_menu.addAction(search\_edit)

interp\_edit = QAction('Interpreter Settings', self)

interp\_edit.setShortcut('Ctrl+I')

interp\_edit.triggered.connect(self.interp\_settings)

interp\_run = QAction('Run Interpreter', self)

interp\_run.setShortcut('F11')

interp\_run.triggered.connect(self.run\_interp)

tools\_menu.addAction(interp\_edit)

tools\_menu.addAction(interp\_run)

self.setMenuWidget(main\_menu)

self.setMenuBar(main\_menu)

@staticmethod

def close\_application():

logger.info('Closing window')

sys.exit()

def add\_tool\_bar(self):

new\_action = QAction(QIcon('pyvicky/staticfiles/plus.png'), 'New item', self)

new\_action.triggered.connect(self.new\_file)

trash\_action = QAction(QIcon('pyvicky/staticfiles/trash.png'), 'Clean item', self)

trash\_action.triggered.connect(self.clear\_text)

run\_action = QAction(QIcon('pyvicky/staticfiles/next.png'), 'Run interpreter', self)

run\_action.triggered.connect(self.run\_interp)

settings\_action = QAction(QIcon('pyvicky/staticfiles/settings.png'), 'Interpreter settings', self)

settings\_action.triggered.connect(self.interp\_settings)

info\_action = QAction(QIcon('pyvicky/staticfiles/information.png'), 'Information', self)

info\_action.triggered.connect(self.about)

self.toolBar = self.addToolBar("Extraction")

self.toolBar.addAction(new\_action)

self.toolBar.addAction(trash\_action)

self.toolBar.addAction(run\_action)

self.toolBar.addAction(settings\_action)

self.toolBar.addAction(info\_action)

def open\_file\_name\_dialog(self):

"""

open 1 file

:return:

"""

options = QFileDialog.Options()

options |= QFileDialog.DontUseNativeDialog

file\_name, \_ = QFileDialog.getOpenFileName(self, "QFileDialog.getOpenFileName()", "",

"All Files (\*);;Python Files (\*.py);;INI files (\*.ini)",

options=options)

if file\_name:

logger.debug(file\_name)

# self.open\_file(file\_name)

return file\_name

return None

def open\_file\_names\_dialog(self):

"""

open several files

:return:

"""

options = QFileDialog.Options()

options |= QFileDialog.DontUseNativeDialog

files, \_ = QFileDialog.getOpenFileNames(self, "QFileDialog.getOpenFileNames()", "",

"All Files (\*);;Python Files (\*.py)", options=options)

if files:

logger.debug(files)

logger.info('Files opened')

return files

def open\_dir\_dialog(self):

"""

open directory

:return:

"""

options = QFileDialog.Options()

options |= QFileDialog.DontUseNativeDialog

dirs = QFileDialog.getExistingDirectory(self, 'Select directory')

if dirs:

logger.debug(dirs)

logger.info('Dir opened')

return dirs

def open\_directory(self):

path = self.open\_dir\_dialog()

if path:

# print(path)

self.treeWidget.initUI(path)

return path

def open\_file(self, filename=None):

try:

if not filename:

filename = self.open\_file\_name\_dialog()

if filename and os.path.isfile(filename):

self.tabWidget.add\_tab(self, filename, open(filename).read())

# self.text.setPlainText(open(filename).read())

self.update\_ui()

logger.info('File opened')

self.set\_title(filename)

except IOError as io\_e:

logger.error(io\_e)

sys.exit(1)

except Exception as e:

logger.error(e)

sys.exit(1)

def open\_files(self):

# TODO: not implemented, implement, when tabs will be created

filename, \_ = QFileDialog.getOpenFileName(self, "Open File", '', "All Files (\*)")

try:

self.tabWidget.add\_tab(self, filename, open(filename).read())

# self.text.setPlainText(open(filename).read())

except IOError as io\_e:

logger.error(io\_e)

sys.exit(1)

except Exception as e:

logger.error(e)

sys.exit(1)

def save\_file\_dialog(self):

"""

save 1 file

:return:

"""

options = QFileDialog.Options()

options |= QFileDialog.DontUseNativeDialog

file\_name, \_ = QFileDialog.getSaveFileName(self, "QFileDialog.getSaveFileName()", "",

"All Files (\*);;Text Files (\*.txt);;Python Files (\*.py)",

options=options)

if file\_name:

logger.debug(file\_name)

# self.save\_file(file\_name)

return file\_name

return None

def save\_file(self, current=None):

try:

file\_name = self.save\_file\_dialog()

with open(file\_name, "w") as CurrentFile:

if not current:

current = self.tabWidget.get\_cur\_index()

CurrentFile.write(self.get\_editor\_by\_index(current).toPlainText())

# CurrentFile.write(self.text.toPlainText())

CurrentFile.close()

self.set\_title(file\_name)

logger.info('File saved')

except IOError as io\_e:

logger.error(io\_e)

sys.exit(1)

except Exception as e:

logger.error(e)

sys.exit(1)

def copy\_func(self):

current = self.tabWidget.get\_cur\_index()

self.get\_editor\_by\_index(current).copy()

def paste\_func(self):

current = self.tabWidget.get\_cur\_index()

self.get\_editor\_by\_index(current).paste()

def get\_editor\_by\_index(self, current):

return self.tabWidget.get\_editor\_from\_tab(current)

def track\_unsaved\_file(self, current=None):

# destroy = self.text.document().isModified()

if current is None:

current = self.tabWidget.get\_cur\_index()

destroy = self.get\_editor\_by\_index(current).document().isModified()

if not destroy:

return False

else:

detour = QMessageBox.question(self,

"Hold your horses.",

"File has unsaved changes. Save now?",

QMessageBox.Yes | QMessageBox.No |

QMessageBox.Cancel)

if detour == QMessageBox.Cancel:

return True

elif detour == QMessageBox.No:

return False

elif detour == QMessageBox.Yes:

return self.save\_file(current)

return True

def track\_unsaved\_files(self):

count = self.tabWidget.get\_tabs()

result = False

for i in range(count):

if not result: # in order to define, is there any file, which is modified and not saved

result = self.track\_unsaved\_file(i)

return result

def closeEvent(self, event):

"""

not a method, but event from QMainWindow

so, its name should be camel - cased

:param event:

:return:

"""

if self.track\_unsaved\_files():

event.ignore()

else:

self.close\_application()

def new\_file(self):

self.tabWidget.add\_tab(self, "new")

# self.clear\_text()

self.update\_ui()

logger.info('File created')

def clear\_text(self):

current = self.tabWidget.get\_cur\_index()

self.get\_editor\_by\_index(current).clear()

def set\_title(self, file\_path):

file\_path = file\_path.split('/')[-1]

self.setWindowTitle('PyVicky: ' + file\_path)

def setup\_editor(self):

# default values

self.load\_config()

# TODO: set font config with editor (or separated file)

self.font.setFamily(self.settings.get('Editor', 'Font', fallback='Courier'))

self.font.setFixedPitch(self.settings.getboolean('Editor', 'FixedPitch', fallback=True))

self.font.setPointSize(self.settings.getint('Editor', 'FontSize', fallback=11))

self.font.setWordSpacing(self.settings.getfloat('Editor', 'WordSpacing', fallback=1.0))

# TODO: set font config with editor (or separated file)

count = self.tabWidget.get\_tabs()

for i in range(count):

highlighter = self.tabWidget.set\_highlighter(i)

text = self.tabWidget.get\_editor\_from\_tab(i)

text.setPalette(highlighter.get\_palette())

text.setTabStopWidth(40)

text.setFont(self.font)

def load\_config(self):

self.settings.read('pyvicky/configs/settings.ini')

self.config.read(self.settings['Editor']['theme'])

def search\_config(self):

find = Find(self)

find.show()

def change\_preferences(self):

try:

# Create the Preferences dialog

dlg = PreferencesDlg(self)

dlg.show()

dlg.accepted.connect(self.update\_ui)

except Exception as e: # If failed to open Preferences dialog, just open the settings file

self.currentFilePath = os.path.join(os.getcwd(), 'pyvicky/configs/settings.ini')

self.currentFileName = 'settings.ini'

self.firstSave = False

with open(self.currentFilePath, 'r') as f:

self.tabWidget.add\_tab(self, "INI", f.read())

# self.text.clear()

# self.text.setPlainText(f.read())

logging.error(e)

traceback.print\_exc()

def interp\_settings(self):

# Create the Interpreter dialog

try:

dlg = InterpreterDlg(self)

except Exception as e:

logging.error(e)

@staticmethod

def run\_interp():

settings = configparser.ConfigParser()

settings.read('pyvicky/configs/interpreter\_settings.ini')

print(settings.sections())

# settings\_arr = settings.sections()[0]

# settings\_arr = settings.options(settings\_arr)

run\_script = ' '.join([settings['Interpreter']['path'],

settings['Interpreter']['file'],

settings['Interpreter']['keys']])

logging.debug("SETTINGS: " + run\_script)

try:

process = subprocess.Popen('echo "Start executing program..."', stdout=subprocess.PIPE, shell=True)

start = time.time()

process = subprocess.call(run\_script, shell=True)

end = time.time()

elapsed\_time = end - start

logging.info('Task is done in ' + str(elapsed\_time))

print('Task is successfully done in {time} sec'.format(time=str(elapsed\_time)))

except Exception as e:

logging.error(e)

def update\_ui(self):

# Reload the config

self.load\_config()

self.setup\_editor()

# self.text.update()

self.update()

logging.info("Updated preferences")

def about(self):

QMessageBox.about(self, "About Syntax PyVicky",

"<p>The <b>PyVicky Editor</b> is a simple text editor "

"which perform simple syntax highlighting by subclassing the "

"QSyntaxHighlighter class and describing highlighting "

"rules using regular expressions.</p>")

def close\_tab(self, current\_index):

if self.track\_unsaved\_file(current\_index):

pass

else:

self.tabWidget.close\_tab(current\_index)

def open\_file\_from\_tree(self, index):

# TODO: iterate if file is already opned, or not??=)

path = self.treeWidget.tree\_function(index)

self.open\_file(path)

class DirectoriesTreeView(QWidget):

def \_\_init\_\_(self, parent, \_dir=None):

super(QWidget, self).\_\_init\_\_(parent)

self.tree = QTreeView()

windowLayout = QVBoxLayout()

windowLayout.addWidget(self.tree)

self.setLayout(windowLayout)

logging.info('create tree widget')

self.initUI(\_dir)

def initUI(self, \_dir=None):

if \_dir:

self.model = QFileSystemModel()

# self.model.setRootPath('')

self.model.setRootPath(\_dir)

# print(self.model.rootPath())

# print(self.model.rootDirectory())

index = self.model.index(self.model.rootPath())

self.tree.setModel(self.model)

self.tree.setRootIndex(index)

self.tree.setAnimated(False)

self.tree.setIndentation(20)

self.tree.setSortingEnabled(True)

# self.tree.resize(640, 480)

self.tree.hideColumn(3) # disable file type, file size etc

self.tree.hideColumn(2)

self.tree.hideColumn(1)

# self.tree.clicked.connect(self.tree\_function)

logging.info('update tree')

self.show()

def tree\_function(self, index):

# print(item)

path = self.sender().model().filePath(index)

logging.info('Tree path is ' + path)

return path

class TabView(QWidget):

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

super(QWidget, self).\_\_init\_\_(parent)

self.layout = QVBoxLayout(self)

# Initialize tab screen

self.tabWidget = QTabWidget()

self.tabWidget.setTabsClosable(True)

# self.tabWidget.tabCloseRequested.connect(self.close\_tab)

# Add tabs to widget

self.layout.addWidget(self.tabWidget)

self.setLayout(self.layout)

def add\_tab(self, parent, tab\_name, text=''):

tab = QWidget()

tab.layout = QVBoxLayout(self)

text\_editor = QCodeEditor(parent)

text\_editor.setPlainText(text)

tab.layout.addWidget(text\_editor)

tab.setLayout(tab.layout)

if '/' in tab\_name:

tab\_name = tab\_name.split('/')[-1]

self.tabWidget.addTab(tab, tab\_name)

logging.info('Added tab')

def close\_tab(self, currentIndex):

# TODO: control save operation

# currentQWidget = self.widget(currentIndex)

# currentQWidget.deleteLater()

self.tabWidget.removeTab(currentIndex)

logging.info('Closed tab')

def get\_cur\_index(self):

return self.tabWidget.currentIndex()

def get\_cur\_text(self):

index = self.get\_cur\_index()

text = self.tabWidget.widget(index).text\_editor.toPlainText()

return text

def get\_editor\_from\_tab(self, index):

tab = self.tabWidget.widget(index)

count = tab.layout.count()

try:

for i in range(count):

if type(tab.layout.itemAt(i).widget()) is QCodeEditor:

return tab.layout.itemAt(i).widget()

except Exception as e:

logging.error(e)

def set\_highlighter(self, index):

editor = self.get\_editor\_from\_tab(index)

self.tabWidget.widget(index).highlighter = PythonHighlighter(editor.document())

return self.tabWidget.widget(index).highlighter

def get\_tabs(self):

return self.tabWidget.count()

**ПРИЛОЖЕНИЕ 2. Код файла preferences.py**

# preferences dialog

from PyQt5.QtCore import QObject, pyqtSignal

from PyQt5.QtWidgets import QVBoxLayout, QGridLayout, QLineEdit, QHBoxLayout, QPushButton, QSpinBox, QCheckBox, \

QComboBox, QLabel, QGroupBox, QColorDialog, QDialog, QDoubleSpinBox, QLabel

from PyQt5.QtGui import QColor

import configparser

import os

import logging

from . import highlighter as hl

class PreferencesDlg(QDialog):

def \_\_init\_\_(self, parent=None):

super(PreferencesDlg, self).\_\_init\_\_(parent)

self.config = None

self.settings = None

self.theme = ''

self.load\_config()

self.make\_widgets()

self.set\_initial\_values()

self.make\_connections()

self.setWindowTitle('Preferences')

self.show()

def load\_config(self):

self.settings = configparser.ConfigParser()

self.settings.read('pyvicky/configs/settings.ini')

try:

self.theme = 'pyvicky/configs/themes/' + self.themeBox.currentText() + '.ini'

except Exception as e:

logging.error(e)

self.theme = ''

if self.theme == '':

self.theme = self.settings.get('Editor', 'theme', fallback='pyvicky/configs/themes/default.ini')

self.config = configparser.ConfigParser()

try:

self.config.read(self.theme)

self.settings['Editor']['theme'] = self.theme

except Exception as e:

logging.error(e)

self.config.read('pyvicky/configs/themes/default.ini') # Use the default color theme

self.settings['Editor']['theme'] = self.theme

def make\_widgets(self):

self.mainLayout = QVBoxLayout()

self.setup\_editor\_widgets()

self.setup\_color\_widgets()

self.setup\_extensions\_widgets()

button\_layout = QHBoxLayout()

self.saveButton = QPushButton('&Save')

self.cancelButton = QPushButton('Cancel')

button\_layout.addWidget(self.saveButton)

button\_layout.addWidget(self.cancelButton)

self.mainLayout.addLayout(button\_layout)

self.setLayout(self.mainLayout)

def make\_connections(self):

# Color Picker buttons

self.bgButton.clicked.connect(self.select\_bg)

self.fgButton.clicked.connect(self.select\_fg)

self.singleButton.clicked.connect(self.select\_single)

self.multiButton.clicked.connect(self.select\_multi)

self.stringButton.clicked.connect(self.select\_string)

self.keywordButton.clicked.connect(self.select\_keyword)

self.functionButton.clicked.connect(self.select\_function)

self.highlightButton.clicked.connect(self.select\_highlight)

self.highlightedTextButton.clicked.connect(self.select\_highlighted\_text)

# Accept/Cancel buttons

self.saveButton.clicked.connect(self.save)

self.cancelButton.clicked.connect(self.cancel)

@staticmethod

def get\_themes():

"""Parse through the theme dir and get the themes names as strings.

Returns a list (of theme names)."""

themes = []

for file in os.listdir('pyvicky/configs/themes/'):

if file.endswith('.ini'):

themes.append(file[:-4])

return themes

@staticmethod

def get\_paths():

"""Parse through the theme dir and get the themes names as strings.

Returns a list (of theme names)."""

themes = []

for file in os.listdir('pyvicky/keywords/'):

if file.endswith('.txt'):

themes.append('pyvicky/keywords/' + file)

return themes

def set\_color\_values(self):

# Get the theme

self.load\_config()

# Set the widget values

self.bgInput.setText(self.config['Colors']['Background'])

self.fgInput.setText(self.config['Colors']['Foreground'])

self.singleInput.setText(self.config['Colors']['SingleLineComment'])

self.multiInput.setText(self.config['Colors']['MultiLineComment'])

self.stringInput.setText(self.config['Colors']['String'])

self.keywordInput.setText(self.config['Colors']['Keyword'])

self.functionInput.setText(self.config['Colors']['Function'])

self.highlightInput.setText(self.config['Colors']['Highlight'])

self.highlightedTextInput.setText(self.config['Colors']['HighlightedText'])

color\_picker = QColor()

color\_picker.setNamedColor('#' + self.config['Colors']['Background'])

self.bgLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_color\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['Foreground'])

self.fgLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_color\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['SingleLineComment'])

self.singleLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_color\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['MultiLineComment'])

self.multiLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_color\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['String'])

self.stringLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_color\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['Keyword'])

self.keywordLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_color\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['Function'])

self.functionLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_color\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['Highlight'])

self.highlightLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_color\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['HighlightedText'])

self.highlightedTextLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_color\_values " + color\_picker.name())

def theme\_selected(self):

# Theme has been selected, now update the other theme setting widgets

# to contain \*that\* theme's colors

self.set\_color\_values()

@staticmethod

def get\_highlighters():

"""Parse through the highlighter.py file and get the highlighter class names as strings.

Returns a list (of class names)."""

highlighters = []

classes = dir(hl)

classes.remove('BaseHighlighter') # Ignore the base class

for highlighter in classes:

if highlighter.endswith('Highlighter'):

highlighters.append(highlighter)

return highlighters

def setup\_editor\_widgets(self):

# Editor widgets

self.fontBox = QLineEdit()

self.fontSize = QSpinBox()

self.fontSize.setRange(4, 72)

self.fixedPitchToggle = QCheckBox('Fixed Pitch')

self.wordSpacing = QDoubleSpinBox()

self.wordSpacing.setRange(0.5, 5.0)

self.useSpaces = QCheckBox('Use Spaces')

self.spacesPerTab = QSpinBox()

self.spacesPerTab.setRange(1, 8)

self.showLineNumbers = QCheckBox('Show Line Numbers')

self.smartIndent = QCheckBox('Smart Indent')

self.themeBox = QComboBox()

themes = self.get\_themes()

self.themeBox.addItems(themes)

self.themeBox.setCurrentIndex(themes.index(self.settings['Editor']['theme'].split('/')[-1][

:-4])) # Set the current item to the the current theme (without the path and .ini part)

self.themeBox.activated.connect(self.theme\_selected)

# Editor Group

self.editorLayout = QGridLayout()

self.editorLayout.addWidget(QLabel('Font:'), 0, 0)

self.editorLayout.addWidget(self.fontBox, 0, 1)

self.editorLayout.addWidget(QLabel('Font Size:'), 1, 0)

self.editorLayout.addWidget(self.fontSize, 1, 1)

self.editorLayout.addWidget(self.fixedPitchToggle, 2, 0, 1, 2)

self.editorLayout.addWidget(QLabel('Word Spacing:'), 3, 0)

self.editorLayout.addWidget(self.wordSpacing, 3, 1)

self.editorLayout.addWidget(self.useSpaces, 4, 0)

self.editorLayout.addWidget(QLabel('Spaces Per Tab:'), 5, 0)

self.editorLayout.addWidget(self.spacesPerTab, 5, 1)

self.editorLayout.addWidget(self.showLineNumbers, 6, 0)

self.editorLayout.addWidget(self.smartIndent, 7, 0)

self.editorLayout.addWidget(QLabel('Theme:'), 8, 0)

self.editorLayout.addWidget(self.themeBox, 8, 1)

self.editorGroupBox = QGroupBox('Editor')

self.editorGroupBox.setLayout(self.editorLayout)

self.mainLayout.addWidget(self.editorGroupBox)

def setup\_color\_widgets(self):

# Color widgets

self.bgInput = QLineEdit()

self.bgLabel = QLabel(' ')

self.bgButton = QPushButton('Choose...')

self.fgInput = QLineEdit()

self.fgButton = QPushButton('Choose...')

self.fgLabel = QLabel(' ')

self.singleInput = QLineEdit()

self.singleButton = QPushButton('Choose...')

self.singleLabel = QLabel(' ')

self.multiInput = QLineEdit()

self.multiButton = QPushButton('Choose...')

self.multiLabel = QLabel(' ')

self.stringInput = QLineEdit()

self.stringButton = QPushButton('Choose...')

self.stringLabel = QLabel(' ')

self.keywordInput = QLineEdit()

self.keywordButton = QPushButton('Choose...')

self.keywordLabel = QLabel(' ')

self.functionInput = QLineEdit()

self.functionButton = QPushButton('Choose...')

self.functionLabel = QLabel(' ')

self.highlightInput = QLineEdit()

self.highlightButton = QPushButton('Choose...')

self.highlightLabel = QLabel(' ')

self.highlightedTextInput = QLineEdit()

self.highlightedTextButton = QPushButton('Choose...')

self.highlightedTextLabel = QLabel(' ')

# Color layout

self.colorLayout = QGridLayout()

self.colorLayout.addWidget(QLabel('Background:'), 0, 0)

self.colorLayout.addWidget(self.bgInput, 0, 1)

self.colorLayout.addWidget(self.bgButton, 0, 3)

self.colorLayout.addWidget(self.bgLabel, 0, 2)

self.colorLayout.addWidget(QLabel('Foreground:'), 1, 0)

self.colorLayout.addWidget(self.fgInput, 1, 1)

self.colorLayout.addWidget(self.fgButton, 1, 3)

self.colorLayout.addWidget(self.fgLabel, 1, 2)

self.colorLayout.addWidget(QLabel('Single Line Comment:'), 2, 0)

self.colorLayout.addWidget(self.singleInput, 2, 1)

self.colorLayout.addWidget(self.singleButton, 2, 3)

self.colorLayout.addWidget(self.singleLabel, 2, 2)

self.colorLayout.addWidget(QLabel('Mult-Line Comment:'), 3, 0)

self.colorLayout.addWidget(self.multiInput, 3, 1)

self.colorLayout.addWidget(self.multiButton, 3, 3)

self.colorLayout.addWidget(self.multiLabel, 3, 2)

self.colorLayout.addWidget(QLabel('String:'), 4, 0)

self.colorLayout.addWidget(self.stringInput, 4, 1)

self.colorLayout.addWidget(self.stringButton, 4, 3)

self.colorLayout.addWidget(self.stringLabel, 4, 2)

self.colorLayout.addWidget(QLabel('Keyword:'), 5, 0)

self.colorLayout.addWidget(self.keywordInput, 5, 1)

self.colorLayout.addWidget(self.keywordButton, 5, 3)

self.colorLayout.addWidget(self.keywordLabel, 5, 2)

self.colorLayout.addWidget(QLabel('Function:'), 6, 0)

self.colorLayout.addWidget(self.functionInput, 6, 1)

self.colorLayout.addWidget(self.functionButton, 6, 3)

self.colorLayout.addWidget(self.functionLabel, 6, 2)

self.colorLayout.addWidget(QLabel('Highlight:'), 7, 0)

self.colorLayout.addWidget(self.highlightInput, 7, 1)

self.colorLayout.addWidget(self.highlightButton, 7, 3)

self.colorLayout.addWidget(self.highlightLabel, 7, 2)

self.colorLayout.addWidget(QLabel('Highlighted Text:'), 8, 0)

self.colorLayout.addWidget(self.highlightedTextInput, 8, 1)

self.colorLayout.addWidget(self.highlightedTextButton, 8, 3)

self.colorLayout.addWidget(self.highlightedTextLabel, 8, 2)

# Color Group

self.colorGroupBox = QGroupBox('Colors')

self.colorGroupBox.setLayout(self.colorLayout)

self.mainLayout.addWidget(self.colorGroupBox)

def setup\_extensions\_widgets(self):

# Extensions Widgets

highlighters = self.get\_highlighters()

self.highlighter = QComboBox()

self.highlighter.addItems(highlighters)

self.highlighter.setCurrentIndex(highlighters.index(self.settings['Extensions'][

'highlighter']))

# Set the current item to the the current theme (without the path and .ini part)

self.dictBox = QComboBox()

dict\_paths = self.get\_paths()

self.dictBox.addItems(dict\_paths)

self.dictBox.setCurrentIndex(dict\_paths.index(self.settings['Dictionary']['path']))

# Extensions Layout

self.extensionsLayout = QGridLayout()

self.extensionsLayout.addWidget(QLabel('Highlighter Class:'))

self.extensionsLayout.addWidget(self.highlighter)

self.extensionsLayout.addWidget(QLabel('Dictionary:'))

self.extensionsLayout.addWidget(self.dictBox)

# Extensions Group

self.extensionsGroupBox = QGroupBox('Extensions')

self.extensionsGroupBox.setLayout(self.extensionsLayout)

self.mainLayout.addWidget(self.extensionsGroupBox)

def set\_initial\_values(self):

self.fontBox.setText(self.settings['Editor']['Font'])

self.fontSize.setValue(self.settings.getint('Editor', 'FontSize'))

self.fixedPitchToggle.setChecked(self.settings.getboolean('Editor', 'FixedPitch'))

self.wordSpacing.setValue(self.settings.getfloat('Editor', 'WordSpacing'))

self.useSpaces.setChecked(self.settings.getboolean('Editor', 'UseSpaces'))

self.spacesPerTab.setValue(self.settings.getint('Editor', 'SpacesPerTab'))

self.showLineNumbers.setChecked(self.settings.getboolean('Editor', 'ShowLineNumbers'))

self.smartIndent.setChecked(self.settings.getboolean('Editor', 'smartindent'))

self.bgInput.setText(self.config['Colors']['Background'])

self.fgInput.setText(self.config['Colors']['Foreground'])

self.singleInput.setText(self.config['Colors']['SingleLineComment'])

self.multiInput.setText(self.config['Colors']['MultiLineComment'])

self.stringInput.setText(self.config['Colors']['String'])

self.keywordInput.setText(self.config['Colors']['Keyword'])

self.functionInput.setText(self.config['Colors']['Function'])

self.highlightInput.setText(self.config['Colors']['Highlight'])

self.highlightedTextInput.setText(self.config['Colors']['HighlightedText'])

color\_picker = QColor()

color\_picker.setNamedColor('#' + self.config['Colors']['Background'])

self.bgLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_initial\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['Foreground'])

self.fgLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_initial\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['SingleLineComment'])

self.singleLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_initial\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['MultiLineComment'])

self.multiLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_initial\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['String'])

self.stringLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_initial\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['Keyword'])

self.keywordLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_initial\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['Function'])

self.functionLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_initial\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['Highlight'])

self.highlightLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_initial\_values " + color\_picker.name())

color\_picker.setNamedColor('#' + self.config['Colors']['HighlightedText'])

self.highlightedTextLabel.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

logging.debug("set\_initial\_values " + color\_picker.name())

highlighters = self.get\_highlighters()

self.highlighter.setCurrentIndex(highlighters.index(self.settings['Extensions'][

'highlighter']))

dicts = self.get\_paths()

self.dictBox.setCurrentIndex(dicts.index(self.settings['Dictionary'][

'path']))

# Set the current item to the the current theme (without the path and .ini part)

def get\_values(self):

"""Reads the values from the preference widgets and

stores them in the self.settings and self.config ConfigParser object."""

# Get correct configs/theme

self.load\_config()

self.settings.set('Editor', 'Font', self.fontBox.text())

self.settings.set('Editor', 'FontSize', str(self.fontSize.value()))

temp = 'yes' if self.fixedPitchToggle.isChecked() else 'no'

self.settings.set('Editor', 'FixedPitch', temp)

self.settings.set('Editor', 'WordSpacing', str(self.wordSpacing.value()))

temp = 'yes' if self.useSpaces.isChecked() else 'no'

self.settings.set('Editor', 'UseSpaces', temp)

self.settings.set('Editor', 'SpacesPerTab', str(self.spacesPerTab.value()))

temp = 'yes' if self.showLineNumbers.isChecked() else 'no'

self.settings.set('Editor', 'ShowLineNumbers', temp)

temp = 'yes' if self.smartIndent.isChecked() else 'no'

self.settings.set('Editor', 'smartindent', temp)

self.settings.set('Editor', 'theme', 'pyvicky/configs/themes/' + self.themeBox.currentText() + '.ini')

self.config.set('Colors', 'Background', self.bgInput.text())

self.config.set('Colors', 'Foreground', self.fgInput.text())

self.config.set('Colors', 'SingleLineComment', self.singleInput.text())

self.config.set('Colors', 'MultiLineComment', self.multiInput.text())

self.config.set('Colors', 'String', self.stringInput.text())

self.config.set('Colors', 'Keyword', self.keywordInput.text())

self.config.set('Colors', 'Function', self.functionInput.text())

self.config.set('Colors', 'Highlight', self.highlightInput.text())

self.config.set('Colors', 'HighlightedText', self.highlightedTextInput.text())

self.settings.set('Extensions', 'Highlighter', self.highlighter.currentText())

self.settings.set('Dictionary', 'path', self.dictBox.currentText())

def make\_color\_dlg(self, line\_edit, color\_label):

color\_dlg = QColorDialog(self)

color\_dlg.setCurrentColor(QColor('#' + line\_edit.text()))

color\_dlg.show()

def update():

color = color\_dlg.currentColor()

rVal, gVal, bVal, aVal = color.getRgb()

# Pad the hex string, if needed

color\_string = ''

if rVal < 0x0F:

color\_string += '0'

color\_string += '%x' % rVal

if gVal < 0x0F:

color\_string += '0'

color\_string += '%x' % gVal

if bVal < 0x0F:

color\_string += '0'

color\_string += '%x' % bVal

line\_edit.setText(color\_string)

color\_picker = QColor()

color\_picker.setNamedColor('#' + color\_string)

logging.debug("make\_color\_dlg " + color\_picker.name())

color\_label.setStyleSheet("QLabel { background-color: %s }" % color\_picker.name())

color\_dlg.accepted.connect(update)

def select\_bg(self):

self.make\_color\_dlg(self.bgInput, self.bgLabel)

def select\_fg(self):

self.make\_color\_dlg(self.fgInput, self.fgLabel)

def select\_single(self):

self.make\_color\_dlg(self.singleInput, self.singleLabel)

def select\_multi(self):

self.make\_color\_dlg(self.multiInput, self.multiLabel)

def select\_string(self):

self.make\_color\_dlg(self.stringInput, self.stringLabel)

def select\_keyword(self):

self.make\_color\_dlg(self.keywordInput, self.keywordLabel)

def select\_function(self):

self.make\_color\_dlg(self.functionInput, self.functionLabel)

def select\_highlight(self):

self.make\_color\_dlg(self.highlightInput, self.highlightLabel)

def select\_highlighted\_text(self):

self.make\_color\_dlg(self.highlightedTextInput, self.highlightedTextLabel)

def save(self):

# Get values into self.config

self.get\_values()

# save the config to file

with open('pyvicky/configs/settings.ini', 'w') as f:

self.settings.write(f)

logging.info('Settings saved')

# save the config to file

with open(self.settings['Editor']['theme'], 'w') as f:

self.config.write(f)

self.accept()

# close the dialog

self.destroy()

def cancel(self):

self.reject()

self.destroy()

# TODO: create font preferences dialog

class FontPreferencesDlg(QDialog):

def \_\_init\_\_(self, parent=None):

super(FontPreferencesDlg, self).\_\_init\_\_(parent)

class Connector(QObject):

# Define a new signal called 'trigger' that has no arguments.

trigger = pyqtSignal()

def connect\_and\_emit\_trigger(self):

# Connect the trigger signal to a slot.

self.trigger.connect(self.handle\_trigger)

# Emit the signal.

self.trigger.emit()

def handle\_trigger(self):

# Show that the slot has been called.

logging.info("trigger signal received")

ПРИЛОЖЕНИЕ 3. Код файла numberbar.py

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

from PyQt5.QtCore import Qt, QRect, QSize

from PyQt5.QtWidgets import QWidget, QPlainTextEdit, QTextEdit, QCompleter

from PyQt5.QtGui import QColor, QPainter, QTextFormat, QTextCursor

import logging

import configparser

import os

class QLineNumberArea(QWidget):

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

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

self.codeEditor = editor

def sizeHint(self):

return QSize(self.editor.lineNumberAreaWidth(), 0)

def paintEvent(self, event):

self.codeEditor.lineNumberAreaPaintEvent(event)

class QCodeEditor(QPlainTextEdit):

def \_\_init\_\_(self, parent=None):

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

self.lineNumberArea = QLineNumberArea(self)

self.blockCountChanged.connect(self.updateLineNumberAreaWidth)

self.updateRequest.connect(self.updateLineNumberArea)

self.cursorPositionChanged.connect(self.highlightCurrentLine)

self.updateLineNumberAreaWidth(0)

self.completer = None

completer = QAutoComplete()

self.setCompleter(completer)

self.moveCursor(QTextCursor.End)

logging.info('Create editor with number bar')

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('9') \* 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()))

# TODO: make this feature optional

def highlightCurrentLine(self):

extraSelections = []

if not self.isReadOnly():

selection = QTextEdit.ExtraSelection()

lineColor = QColor(Qt.yellow).lighter(160) # change it and make more transparent

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()

# Just to make sure I use the right font

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

def setCompleter(self, completer):

if self.completer:

self.disconnect(self.completer, 0, self, 0)

if not completer:

return

completer.setWidget(self)

completer.setCompletionMode(QCompleter.PopupCompletion)

completer.setCaseSensitivity(Qt.CaseInsensitive)

self.completer = completer

self.completer.activated.connect(self.insertCompletion)

def insertCompletion(self, completion):

tc = self.textCursor()

extra = (len(completion) -

len(self.completer.completionPrefix()))

tc.movePosition(QTextCursor.Left)

tc.movePosition(QTextCursor.EndOfWord)

tc.insertText(completion[len(completion) - extra:len(completion)])

self.setTextCursor(tc)

def textUnderCursor(self):

tc = self.textCursor()

tc.select(QTextCursor.WordUnderCursor)

return tc.selectedText()

def focusInEvent(self, event):

if self.completer:

self.completer.setWidget(self)

QPlainTextEdit.focusInEvent(self, event)

def keyPressEvent(self, event):

if self.completer and self.completer.popup().isVisible():

if event.key() in (

Qt.Key\_Enter,

Qt.Key\_Return,

Qt.Key\_Escape,

Qt.Key\_Tab,

Qt.Key\_Backtab):

event.ignore()

return

isShortcut = (event.modifiers() == Qt.ControlModifier and

event.key() == Qt.Key\_E)

if not self.completer or not isShortcut:

QPlainTextEdit.keyPressEvent(self, event)

ctrlOrShift = event.modifiers() in (Qt.ControlModifier,

Qt.ShiftModifier)

if ctrlOrShift and event.text() is '':

# ctrl or shift key on it's own

return

eow = str("~!@#$%^&\*()\_+{}|:\"<>?,./;'[]\\-=") # end of word

hasModifier = ((event.modifiers() != Qt.NoModifier) and

not ctrlOrShift)

completionPrefix = self.textUnderCursor()

if (not isShortcut and (hasModifier or event.text() is '' or

len(completionPrefix) < 2 or

(event.text()[-1]) in eow)):

self.completer.popup().hide()

return

if completionPrefix != self.completer.completionPrefix():

self.completer.setCompletionPrefix(completionPrefix)

popup = self.completer.popup()

popup.setCurrentIndex(

self.completer.completionModel().index(0, 0))

cr = self.cursorRect()

cr.setWidth(self.completer.popup().sizeHintForColumn(0)

+ self.completer.popup().verticalScrollBar().sizeHint().width())

self.completer.complete(cr)

class QAutoComplete(QCompleter):

# TODO: after setting new keyword setting you need to reload program to get new dictionary

def \_\_init\_\_(self, parent=None):

words = []

self.settings = configparser.ConfigParser()

self.settings.read('pyvicky/configs/settings.ini')

try:

str\_path = self.settings['Dictionary']['path']

with open(str\_path, "r") as f:

for word in f:

words.append(word.strip())

f.close()

except IOError as e:

logging.error("unable to load keyword dictionary")

logging.error(e)

logging.error(os.getcwd())

try:

f = open("/usr/share/dict/words", "r") # default system dict

for word in f:

words.append(word.strip())

f.close()

except IOError:

logging.error("dictionary not in anticipated location")

QCompleter.\_\_init\_\_(self, words, parent)

**ПРИЛОЖЕНИЕ 4. Код файла interpreter\_dialog.py**

from PyQt5.QtCore import QObject, pyqtSignal

from PyQt5.QtWidgets import QVBoxLayout, QGridLayout, QLineEdit, QHBoxLayout, QPushButton, QSpinBox, QCheckBox, \

QComboBox, QLabel, QGroupBox, QColorDialog, QDialog, QDoubleSpinBox, QLabel, QFileDialog

from PyQt5.QtGui import QColor

import configparser

import os

import sys

import platform

import logging

class InterpreterDlg(QDialog):

def \_\_init\_\_(self, parent=None):

super(InterpreterDlg, self).\_\_init\_\_(parent)

self.resize(450, 200)

self.config = None

self.settings = None

self.path = None

self.file = None

self.keys = None

self.load\_config()

self.setWindowTitle('InterpreterDlg')

self.initUi()

logging.info('open interpreter dialog')

self.show()

def initUi(self):

interp\_list = self.search\_available\_interpreter()

self.layout = QGridLayout()

self.browse\_button = QPushButton("Browse", self)

self.browse\_button.clicked.connect(self.apply\_new\_interp)

self.browse\_file\_button = QPushButton("Browse", self)

self.browse\_file\_button.clicked.connect(self.apply\_new\_filepath)

self.interpreter\_edit = QLineEdit(self)

self.file\_path\_edit = QLineEdit(self)

self.keys\_edit = QLineEdit(self)

self.interpreter\_label = QLabel('Interpreter', self)

self.file\_path\_label = QLabel('File path', self)

self.keys\_label = QLabel('Keys', self)

self.ok = QPushButton('Ok', self)

self.ok.clicked.connect(self.accept)

self.cancel = QPushButton('Cancel', self)

self.cancel.clicked.connect(self.decline)

self.layout.addWidget(self.interpreter\_label, 0, 1)

self.layout.addWidget(self.interpreter\_edit, 0, 2)

self.layout.addWidget(self.browse\_button, 0, 3)

self.layout.addWidget(self.file\_path\_label, 1, 1)

self.layout.addWidget(self.file\_path\_edit, 1, 2)

self.layout.addWidget(self.browse\_file\_button, 1, 3)

self.layout.addWidget(self.keys\_label, 2, 1)

self.layout.addWidget(self.keys\_edit, 2, 2)

self.layout.addWidget(self.ok, 3, 1)

self.layout.addWidget(self.cancel, 3, 2)

self.setLayout(self.layout)

self.interpreter\_edit.setText(self.path)

self.file\_path\_edit.setText(self.file)

self.keys\_edit.setText(self.keys)

def apply\_new\_interp(self):

interp = self.open\_file\_name\_dialog()

self.interpreter\_edit.setText(interp)

def apply\_new\_filepath(self):

path = self.open\_file\_name\_dialog()

self.file\_path\_edit.setText(path)

def load\_config(self):

self.settings = configparser.ConfigParser()

self.settings.read('pyvicky/configs/interpreter\_settings.ini')

try:

self.path = self.settings['Interpreter']['path']

self.file = self.settings['Interpreter']['file']

self.keys = self.settings['Interpreter']['keys']

except Exception as e:

logging.error(e)

self.config.read('pyvicky/configs/settings.ini') # Use the default color theme

self.path = self.config['Interpreter']['path']

self.file = self.config['Interpreter']['file']

self.keys = self.config['Interpreter']['keys']

def change\_config(self):

self.settings['Interpreter']['path'] = self.interpreter\_edit.text()

self.settings['Interpreter']['file'] = self.file\_path\_edit.text()

self.settings['Interpreter']['keys'] = self.keys\_edit.text()

# save the config to file

with open('pyvicky/configs/interpreter\_settings.ini', 'w') as f:

self.settings.write(f)

logging.info('Settings saved')

@staticmethod

def search\_available\_interpreter():

# current\_platform = platform.platform()

system = platform.system()

if system == 'Linux':

try:

return str(os.popen('whereis python').read()).split(' ')

except Exception as e:

logging.error(e)

elif system == 'Windows':

return [sys.executable]

else:

logging.error('system is not supported')

return None

def open\_file\_name\_dialog(self):

"""

open 1 file

:return:

"""

options = QFileDialog.Options()

options |= QFileDialog.DontUseNativeDialog

file\_name, \_ = QFileDialog.getOpenFileName(self, "Get interpreter", "",

"All Files (\*)",

options=options)

if file\_name:

logging.debug(file\_name)

return file\_name

def accept(self):

self.change\_config()

self.destroy()

def decline(self):

self.reject()

self.destroy()

ПРИЛОЖЕНИЕ 5. Код файла highlighter.py

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

from PyQt5 import QtGui, QtCore

from PyQt5.QtWidgets import QErrorMessage

import logging

import configparser

class BaseHighlighter(QtGui.QSyntaxHighlighter):

def \_\_init\_\_(self, parent=None):

super(BaseHighlighter, self).\_\_init\_\_(parent)

self.parent = parent

self.indenters = []

self.dedenters = []

def get\_palette(self):

raise NotImplementedError('You must subclass BaseHighlighter and implement this function.'

' It should return a QtGui.QPalette.')

class PythonHighlighter(BaseHighlighter):

def \_\_init\_\_(self, parent=None):

super(PythonHighlighter, self).\_\_init\_\_(parent)

self.parent = parent

self.load\_config()

self.classRegex = QtCore.QRegExp("\\bQ[A-Za-z]+\\b")

self.singleLineCommentRegex = QtCore.QRegExp("#[^\n]\*")

self.multiLineCommentRegex = None

self.singleQuoteRegex = QtCore.QRegExp("'.\*'")

self.doubleQuoteRegex = QtCore.QRegExp("\".\*\"")

self.functionRegex = QtCore.QRegExp("\\b[A-Za-z0-9\_]+(?=\\()")

self.commentStartRegex = QtCore.QRegExp('/"""')

self.commentEndRegex = QtCore.QRegExp('"""/')

self.commentChar = '#'

keyword\_format = QtGui.QTextCharFormat()

keyword\_format.setFontWeight(QtGui.QFont.Bold)

self.keywordPatterns = self.get\_patterns('pyvicky/keywords/python\_keywords.txt')

self.highlightingRules = [(QtCore.QRegExp(pattern), keyword\_format)

for pattern in self.keywordPatterns]

class\_format = QtGui.QTextCharFormat()

class\_format.setFontWeight(QtGui.QFont.Bold)

self.highlightingRules.append((self.classRegex,

class\_format))

single\_line\_comment\_format = QtGui.QTextCharFormat()

self.highlightingRules.append((self.singleLineCommentRegex, single\_line\_comment\_format))

self.multiLineCommentFormat = QtGui.QTextCharFormat()

quotation\_format = QtGui.QTextCharFormat()

self.highlightingRules.append((self.doubleQuoteRegex,

quotation\_format))

self.highlightingRules.append((self.singleQuoteRegex,

quotation\_format))

function\_format = QtGui.QTextCharFormat()

function\_format.setFontItalic(True)

function\_format.setForeground(QtCore.Qt.blue)

self.highlightingRules.append((self.functionRegex,

function\_format))

self.commentStartExpression = self.commentStartRegex

self.commentEndExpression = self.commentEndRegex

# For use by the main editor

self.indenters = ['def', 'for', 'while', 'do', 'class', 'if', 'else', 'elif', 'switch', 'case', 'try',

'except', 'finally']

self.dedenters = ['break', 'continue', 'return']

def highlightBlock(self, text):

for pattern, \_format in self.highlightingRules:

expression = QtCore.QRegExp(pattern)

index = expression.indexIn(text)

while index >= 0:

length = expression.matchedLength()

self.setFormat(index, length, \_format)

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

self.setCurrentBlockState(0)

start\_index = 0

if self.previousBlockState() != 1:

start\_index = self.commentStartExpression.indexIn(text)

while start\_index >= 0:

end\_index = self.commentEndExpression.indexIn(text, start\_index)

if end\_index == -1:

self.setCurrentBlockState(1)

comment\_length = len(text) - start\_index

else:

comment\_length = end\_index - start\_index + self.commentEndExpression.matchedLength()

self.setFormat(start\_index, comment\_length, self.multiLineCommentFormat)

start\_index = self.commentStartExpression.indexIn(text, start\_index + comment\_length)

@staticmethod

def get\_patterns(file\_path):

patterns = []

with open(file\_path, 'r') as f:

for keyword in f:

patterns.append('\\b{}\\b'.format(keyword.strip()))

return patterns

def load\_config(self):

self.settings = configparser.ConfigParser()

self.settings.read('pyvicky/configs/settings.ini')

self.theme = configparser.ConfigParser() # The theme config

try:

self.theme.read(self.settings['Editor']['theme'])

except Exception as e:

logging.error(e)

self.theme.read('pyvicky/configs/themes/default.ini')

self.make\_error\_popup(msg='Unable to load theme at path specified in settings.ini. '

'Reverting to default theme.')

def get\_QColor(self, color\_string):

try:

# Assume the string is hexadecimal RGB

rVal = int(color\_string[:2], 16)

gVal = int(color\_string[2:4], 16)

bVal = int(color\_string[4:6], 16)

return QtGui.QColor(rVal, gVal, bVal)

except Exception as e:

logging.error(e)

# Not hex string, so try a built-in QColor color

try:

return QtGui.QColor('#' + color\_string)

except Exception as e:

logging.error(e)

return None

def get\_palette(self):

palette = QtGui.QPalette()

# Keywords

keyword\_color = self.get\_QColor(self.theme['Colors']['Keyword'])

if keyword\_color:

keyword\_format = QtGui.QTextCharFormat()

keyword\_format.setForeground(keyword\_color)

self.highlightingRules += [(pattern, keyword\_format) for pattern in self.keywordPatterns]

else:

self.make\_error\_popup(msg='Unable to load the color for Keyword from settings')

# Background Color

bg\_color = self.get\_QColor(self.theme['Colors']['Background'])

if bg\_color:

palette.setColor(QtGui.QPalette.Base, bg\_color)

else:

self.make\_error\_popup(msg='Unable to load the color for Background from settings')

# Foreground Color

fgc = self.get\_QColor(self.theme['Colors']['Foreground'])

if fgc:

palette.setColor(QtGui.QPalette.Text, fgc)

else:

self.make\_error\_popup(msg='Unable to load the color for Foreground from settings')

# Single Line Comments

line\_comment\_color = self.get\_QColor(self.theme['Colors']['SingleLineComment'])

if line\_comment\_color:

line\_comment\_format = QtGui.QTextCharFormat()

line\_comment\_format.setForeground(line\_comment\_color)

self.highlightingRules.append((self.singleLineCommentRegex, line\_comment\_format))

else:

self.make\_error\_popup(msg='Unable to load the color for SingleLineComment from settings')

# Single Quotes

single\_quote\_color = self.get\_QColor(self.theme['Colors']['String'])

if single\_quote\_color:

single\_quote\_format = QtGui.QTextCharFormat()

single\_quote\_format.setForeground(single\_quote\_color)

self.highlightingRules.append((self.singleQuoteRegex, single\_quote\_format))

else:

self.make\_error\_popup(msg='Unable to load the color for String from settings')

# Double Quotes (uses same as single quote string)

if single\_quote\_color:

single\_quote\_format = QtGui.QTextCharFormat()

single\_quote\_format.setForeground(single\_quote\_color)

self.highlightingRules.append((self.doubleQuoteRegex, single\_quote\_format))

# Functions

function\_color = self.get\_QColor(self.theme['Colors']['Function'])

if function\_color:

function\_format = QtGui.QTextCharFormat()

function\_format.setForeground(function\_color)

self.highlightingRules.append((self.functionRegex, function\_format))

else:

self.make\_error\_popup(msg='Unable to load the color for Function from settings')

select\_color = self.get\_QColor(self.theme['Colors']['Highlight'])

if select\_color:

palette.setColor(QtGui.QPalette.Highlight, select\_color)

else:

self.make\_error\_popup(msg='Unable to load the color for Highlight from settings')

selected\_text\_color = self.get\_QColor(self.theme['Colors']['HighlightedText'])

if selected\_text\_color:

palette.setColor(QtGui.QPalette.HighlightedText, selected\_text\_color)

else:

self.make\_error\_popup(msg='Unable to load the color for HighlightedText from settings')

return palette

@staticmethod

def make\_error\_popup(title='Oops', msg='Something went wrong...'):

popup = QErrorMessage()

popup.setWindowTitle(title)

popup.showMessage(msg)

class CppHighlighter(BaseHighlighter):

def get\_palette(self):

pass

ПРИЛОЖЕНИЕ 6. Код файла find.py

from PyQt5.QtWidgets import QWidget, QPlainTextEdit, QTextEdit, QCompleter, QDialog, QPushButton, QRadioButton, \

QLineEdit, QLabel, QCheckBox, QGridLayout

from PyQt5.QtGui import QTextFormat, QTextCursor

import re

import logging

class Find(QDialog):

def \_\_init\_\_(self, parent=None):

QDialog.\_\_init\_\_(self, parent)

self.parent = parent

self.lastMatch = None

self.initUI()

self.findField.setFocus()

logging.info('created find window')

def initUI(self):

# Search button

findButton = QPushButton('&Find', self)

findButton.clicked.connect(self.find)

# findButton.returnPressed.connect(self.find)

# Replace Button

replaceButton = QPushButton('&Replace', self)

replaceButton.clicked.connect(self.replace)

# replaceButton.returnPressed.connect(self.replace)

# Replace All Button

replaceAllButton = QPushButton('Replace &All', self)

replaceAllButton.clicked.connect(self.replaceAll)

# Normal Mode

self.normalRadio = QRadioButton('Normal', self)

self.normalRadio.toggled.connect(self.normalMode)

# Regex Mode

self.regexRadio = QRadioButton('RegEx', self)

self.regexRadio.toggled.connect(self.regexMode)

# Search input

self.findField = QLineEdit(self)

# Replace input

self.replaceField = QLineEdit(self)

self.replaceField.returnPressed.connect(self.replace)

optionsLabel = QLabel('Options: ', self)

# Case sensitivity option

self.caseSensitive = QCheckBox('Case sensitive', self)

# Whole words option

self.wholeWords = QCheckBox('Whole words', self)

# Layout

layout = QGridLayout()

layout.addWidget(self.findField, 1, 0, 1, 4)

layout.addWidget(self.normalRadio, 2, 2)

layout.addWidget(self.regexRadio, 2, 3)

layout.addWidget(findButton, 2, 0, 1, 2)

layout.addWidget(self.replaceField, 3, 0, 1, 4)

layout.addWidget(replaceButton, 4, 0, 1, 2)

layout.addWidget(replaceAllButton, 4, 2, 1, 2)

spacer = QWidget(self)

spacer.setFixedSize(0, 10)

layout.addWidget(spacer, 5, 0)

layout.addWidget(optionsLabel, 6, 0)

layout.addWidget(self.caseSensitive, 6, 1)

layout.addWidget(self.wholeWords, 6, 2)

self.setGeometry(300, 300, 360, 250)

self.setWindowTitle('Find and Replace')

self.setLayout(layout)

# Normal mode by default

self.normalRadio.setChecked(True)

logging.info('Initialized find ui')

def get\_parent\_text(self):

current\_index = self.parent.tabWidget.get\_cur\_index()

return self.parent.tabWidget.get\_editor\_from\_tab(current\_index)

def find(self):

# Get text

text = self.get\_parent\_text().toPlainText()

# And text to Find

query = self.findField.text()

if self.wholeWords.isChecked():

query = r'\W' + query + r'\W'

# By default regexes are case sensitive but usually a search isn't

flags = 0 if self.caseSensitive.isChecked() else re.I

# Compile the pattern

pattern = re.compile(query, flags)

# use last match position, if available

start = self.lastMatch.start() + 1 if self.lastMatch else 0

# The actual search

self.lastMatch = pattern.search(text, start)

if self.lastMatch:

start = self.lastMatch.start()

end = self.lastMatch.end()

# If using whole words, include the two non-alphanumerics wrapping it

if self.wholeWords.isChecked():

start += 1

end -= 1

self.moveCursor(start, end)

else:

# set the cursor to the end if search failed

self.get\_parent\_text().moveCursor(QTextCursor.End)

def replace(self):

# Get cursor

cursor = self.get\_parent\_text().textCursor()

# Security

if self.lastMatch and cursor.hasSelection():

# Insert the new text, which will override the selected text

cursor.insertText(self.replaceField.text())

# And set the new cursor

self.get\_parent\_text().setTextCursor(cursor)

def replaceAll(self):

# Set lastmatch to None so the search starts from the beginning

self.lastMatch = None

# Initial find() call so that lastMatch is potentially not None

self.find()

# Replace and find until find is None again

while self.lastMatch:

self.replace()

self.find()

def regexMode(self):

# First uncheck the checkboxes

self.caseSensitive.setChecked(False)

self.wholeWords.setChecked(False)

# Then disable them

self.caseSensitive.setEnabled(False)

self.wholeWords.setEnabled(False)

def normalMode(self):

# Enable checkboxes

self.caseSensitive.setEnabled(True)

self.wholeWords.setEnabled(True)

def moveCursor(self, start, end):

# Get the QTextCursor object from parent's QTextEdit

cursor = self.get\_parent\_text().textCursor()

# Then set position to beginning of last match

cursor.setPosition(start)

# Move the Cursor over the match and pass KeepAnchor to select the text

cursor.movePosition(QTextCursor.Right, QTextCursor.KeepAnchor, end - start)

# And finally, set this new cursor as the parent's

self.get\_parent\_text().setTextCursor(cursor)