UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Information Technology

Taavi Ilp

Improving the Usability of the Thonny Integrated Development Environment

Bachelor’s Thesis (6 ECTS)

Supervisor: Aivar Annamaa, BA

Tartu 2015

Improving the Usability of the Thonny Integrated Development Environment

Summary:

The thesis contains a description of a software development project that was launched to improve the usability of Thonny, an integrated Python development environment designed for novice programmers. First, the concept of usability of beginners' IDE is examined based on a literature study. The main findings of an expert analysis of the usability of Thonny are then presented. The thesis gives a detailed overview of the new features and improvements that were implemented and integrated with Thonny based on the results of the analysis. In addition, a number of ideas for future Thonny features and implementation changes are listed.

Keywords:

Python, Programming, IDE, Usability

Thonny arenduskeskkonna kasutatavuse täiustamine

Lühikokkuvõte:

Bakalaureusetöö kirjeldab autori poolt teostatud tarkvaraarendusprojekti, mille eesmärgiks oli Pythoni programmeerimiskeele algõppeks loodud arenduskeskkonna Thonny täiustamine. Esmalt uuritakse erialase kirjanduse põhjal kasutatavuse mõistet algajatele programmeerijatele mõeldud arenduskeskkonna kontekstis. Seejärel tuuakse välja Thonny kasutatavuse ekspertanalüüsi peamised tulemused. Kirjeldatakse ka tarkvaraarenduse käigus valminud uusi funktsionaalseid võimalusi ning selgitatakse nende positiivset panust Thonny kasutatavuse aspektist vaadelduna. Samuti on bakalaureusetöös välja toodud mõned autori ideed Thonny edasiste võimalike arenduste osas.

Võtmesõnad:

Python, Programmeerimine, IDE, Arenduskeskkond, Kasutatavus

Table of Contents

Introduction 5
Usability and Programming Education 6
General Concept of Usability 6
Difficulties of Learning Introductory Programming 6
Overview of Usability of Beginners' IDE 7
Thonny Integrated Development Environment 10
Overview of Thonny 10
Thonny's Graphical User Interface 10
Editors Notebook 11
Editor 12
Menu 12
Shell 12
File Browser 13
Views Pane 13
Analysis of Thonny's Usability 14
Methodology 14
Results of the Analysis 15
Final Project Scope 17
Development and Deliverables 19
Development Principles 19
Find & Replace Window 20
Implementation Overview 20
Predicted Usability Improvements 22
Suggested Future Work 22
Autocomplete Functionality 23

Implementation Overview 23
Predicted Usability Improvement 25
Outline View 25
Implementation Overview 25
Predicted Usability Improvement 27
Suggested Future Work 27
Identifier Refactoring Support 28
Implementation Overview 28
Predicted Usability Improvement 29
Suggested Future Work 29
Block Comment Toggle 30
Implementation Overview 30
Predicted Usability Improvement 30
Future Thonny Development Suggestions 31
Conclusion 33
References 34
Appendix A: Test Suites 37
Appendix B: License 66

Introduction

The thesis describes a software development project that was completed to improve the usability of Thonny, an integrated development environment (IDE) that is mainly intended to be used by university students taking an introductory programming course.

Based on a literature study in the fields of usability and computer science education, the thesis will first examine the general concept of usability, followed by an analysis of the unique needs and problems of novice programmers. Subsequently, the relationship between the usability of an IDE and learning introductory programming will be examined. It will be demonstrated that a user-friendly and intuitive programming environment is conducive to increased learning productivity, and that various unique factors must be considered when designing and developing such IDEs compared to general purpose IDEs.

The thesis will then describe the Thonny integrated Python development environment, which has been designed and developed by Aivar Annamaa at the University of Tartu. Although Thonny can be used as a general purpose lightweight IDE similar to IDLE, it has been created to be mainly used for the purposes of teaching and learning introductory programming, and thus provides various tools to assist in a pedagogical environment. An expert analysis of the usability of Thonny will be presented, concluding with the list of new features that were selected for development and integration with Thonny to address some of the identified usability concerns.

An overview of the development project will then follow, with detailed descriptions of the final implementations of the features with short usage guides and illustrating screenshots. Finally, author’s ideas and suggestions for future developments which would further improve Thonny will be listed.

The complete list of test suites that were used to verify the quality of the delivered code can be found in the Appendix A of the thesis.

Usability and Programming Education

Before Thonny's usability analysis was started and the scope of the development project finalized, a thorough literature study was performed in the fields of usability and computer science education. The following chapter summarizes the main findings of the literature study on the general concept of usability, the various challenges faced by novice programmers and the attributes considered beneficial for a beginners' IDE.

General Concept of Usability

A general definition of usability, as formulated by Nielsen, is as follows: "Usability is a quality attribute that assesses how easy user interfaces are to use" [1]. Nielsen contrasts the usability quality attribute with utility, which assesses whether a software program provides the features needed by the user [1]. The same distinction is made in the current thesis.

An important fact to consider when discussing a program's usability is that it should always be evaluated from the perspective of the intended end users of the software [2]. Therefore, decisions regarding usability should take into account the specific needs and prior experiences of the users. For example, a professional programmer would very likely have significantly different usability requirements from the development tools than a student who is beginning to learn programming [3]. Thus the problems faced by novice programmers had to be broadly understood before the usability priorities of a beginners' IDE were formulated.

Difficulties of Learning Introductory Programming

The learning difficulties that novice programmers often face are widely discussed in literature [4, 5], with many students failing to acquire sufficient programming knowledge and skills during the introductory programming courses. This has been a cause of major concern for decades as these courses teach students the material that is absolutely vital for subsequent courses in computer science programs [6].

The reasons for failing to acquire the course material at a satisfactory level are likely to vary considerably by individual students. A more detailed analysis is beyond the scope of the current thesis, but some of the common problems that novice programmers often face are the following:

- Failure to apply acquired knowledge to solve problems [4];
- Inability to approach code generation from a sufficiently top-down abstract level [3, 4, 7];
- Failure to efficiently trace the cause of unpredicted program behaviour [4, 8];
- Surface-level knowledge of syntax [4];
- Incomplete or incorrect mental model of code execution [3, 9];
- Inefficient organization of cognitive working memory [4, 7];
- Unwillingness to refactor or modify their program upon discovering implementation mistakes [9, 10].

Additionally, it is very likely that the majority of students have never used an IDE prior to their first programming course. Therefore, learning to independently use their IDE constitutes yet another challenge that students must overcome.

Overview of Usability of Beginners' IDE

The usability of a software program depends on the expected needs and usage patterns of its end users. It serves the pragmatic purpose of assisting its users to achieve their goals, which in the case of a beginners' IDE means providing its users with tools to focus more efficiently on acquiring the relevant programming knowledge and practical skills. The presence of tools which assist with eliminating time-consuming but trivial tasks helps to save students' time and mental energy, which could instead better be used for abstract reasoning and meaningful code generation [10, 11]. Time spent for independent code generation correlates well with positive results [4, 12, 13] and has been shown to increase student confidence in their skills [6, 11]. High usability also eliminates some unnecessary causes of frustration, which may also have a positive effect on their learning progress as students' emotional response to programming has been shown to be significant for successfully acquiring the introductory programming course material [4, 5, 9].

A distinctive attribute of a beginners' IDE is that the demands of its users change quite rapidly as the students acquire more programming knowledge [4]. During the initial IDE usages, the presence of a large number of features that students at that point in their programming education do not comprehend or need - the so-called 'feature clutter' - is highly likely to confuse and overwhelm beginners [3, 14, 15]. It therefore seems reasonable to limit the number of IDE features available or at least visible to students to those they need and understand, improving the learnability component of usability [1, 11]. The more advanced features can then be enabled and introduced later on in the course, when students are already more comfortable with the IDE [14].

Whenever possible and feasible, the features that novices are initially introduced to at the beginning of the course should be similar to analogous functions in software programs that the students are likely to be already familiar with [7, 10, 13]. For example, university students have likely used various text editors for completing their high school assignments. It would therefore facilitate their learning process and increase their confidence if they were able to use familiar tools for some aspects of code generation [16].

As the students' programming knowledge increases, so does the size of the source code they generate. Some of the more curious students are also likely to start expanding their programming knowledge on their own, for example by writing simple computer games or investigating the source code of programs they are interested in [13]. Therefore they start needing tools which help to deal with the code complexity, for example by providing quick navigation assistance or helping to perform multiple related changes together. Students might also find it useful to have access to graphical overviews providing information on the current code structure, such as a display of its class and method tree [10]. Such additional tools improve the IDE usability by making code generation, refactoring and error tracing simpler and faster [12, 15], directly leading to more learning opportunities [13].

Finally, it should be taken into account that the IDE's non-functional attributes such as its performance and stability can have a significant effect on its perceived usability [15]. The ability of beginners to download, install and start using the IDE on their own is a significant contributor to its usability [7, 14]. Students would also likely find it very distracting and frustrating to deal with slowness or unresponsiveness, frequent unexpected error messages, component failures or even program crashes [15, 17]. Furthermore, IDE stability issues during a graded test or an exam may cause students to lose their work, creating educational complications. Therefore, the IDE and its features should be thoroughly and consistently tested and maintained to ensure long-term and high quality performance and stability.

The current chapter summarized the main findings of the literature study which investigated the concept of usability within the context of an IDE used mainly by introductory programming course students. This information formed the theoretical background which the subsequent Thonny's usability analysis was based upon.

Thonny Integrated Development Environment

The following chapter gives an overview of the Thonny integrated development environment. First, a general overview of Thonny is given, including its history and the non-functional attributes. This is followed by a description of Thonny's graphical user interface and the relevant details of its main components.

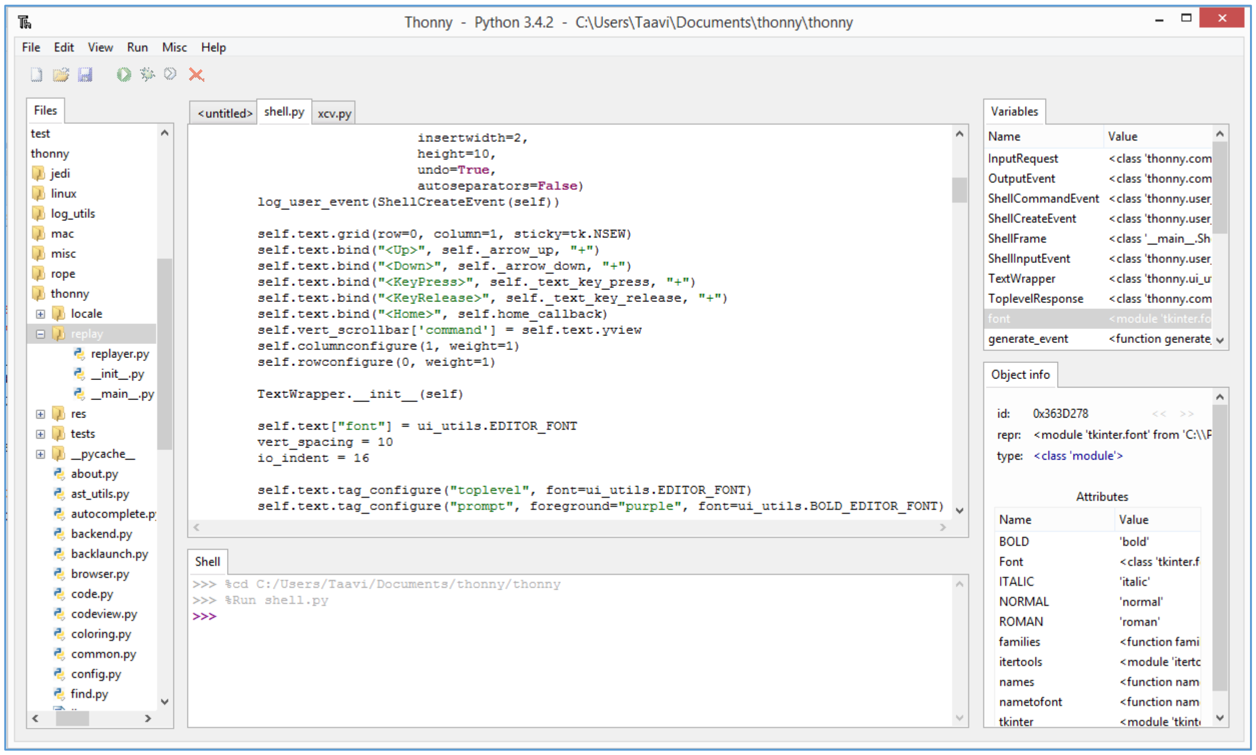
Overview of Thonny

Thonny is an integrated Python development environment, which has been designed and developed by Aivar Annamaa at the University of Tartu. It is sufficiently mature in its development to have been used by some students for the introductory programming course at the University of Tartu as a voluntary alternative to IDLE, the IDE bundled with the default implementation of Python [18]. It provides a beginner-friendly development environment suitable for novice programming students, while also offering many of the features and options expected from a modern IDE. A significant amount of the development focus thus far has been on implementing debugging tools accessible for beginners, such as an interface for stepping through the code as it is executed and runtime object information inspection. Thonny is open-source, free to use and plans are in place for its active future development and support [17]. Windows, Linux and Mac OS distributions are available for each release version and its only dependencies are Python 3.2 or later along with its standard libraries. Any Python 3 programs can be compiled and launched with Thonny. Currently, the only language available for its graphical user interface is English, although support for other language packs may be added in the future. Source code and binary release executable can be downloaded from a public BitBucket repository [19].

Thonny's Graphical User Interface

Although the full overview of Thonny is out of the scope of the current thesis, the following subchapter gives an overview of the graphical user interface of Thonny, describing its main layout and listing its most important components. Due to the fact that no relevant documentation or program-level help exist, the presented information is based on the impressions and understanding of the author of the current thesis. The names of all the interface elements are also derived by the author of the current thesis based on source code variable names and common naming conventions of graphical interface components.

The graphical user interface of Thonny uses just one top level window, which acts as a container for the individual sub-components and areas. The size of the Thonny window can be adjusted by the user, with its components attempting to correspondingly resize themselves as the main window is resized. The size of many of the individual components can also be changed in relation to each other. Thonny's graphical user interface is implemented in Python 3 and extensively uses the TkInter framework.



```
                                 insertwidth=2,
                                 height=10,
                                 undo=True,
                                 autoseparators=False)
        log_user_event(ShellCreateEvent(self))

        self.text.grid(row=0, column=1, sticky=tk.NSEW)
        self.text.bind("<Up>", self._arrow_up, "+")
        self.text.bind("<Down>", self._arrow_down, "+")
        self.text.bind("<KeyPress>", self._text_key_press, "+")
        self.text.bind("<KeyRelease>", self._text_key_release, "+")
        self.text.bind("<Home>", self.home_callback)
        self.vert_scrollbar['command'] = self.text.yview
        self.columnconfigure(1, weight=1)
        self.rowconfigure(0, weight=1)

        TextWrapper.__init__(self)

        self.text["font"] = ui_utils.EDITOR_FONT
        vert_spacing = 10
        io_indent = 16

        self.text.tag_configure("toplevel", font=ui_utils.EDITOR_FONT)
        self.text.tag_configure("prompt", foreground="purple", font=ui_utils.BOLD_EDITOR_FONT)
```

```
>>> %cd C:/Users/Taavi/Documents/thonny/thonny
>>> %Run shell.py
>>>
```

Name	Value
InputRequest	<class 'thonny.com
OutputEvent	<class 'thonny.com
ShellCommandEvent	<class 'thonny.user
ShellCreateEvent	<class 'thonny.user
ShellFrame	<class '__main__.Sh
ShellInputEvent	<class 'thonny.user
TextWrapper	<class 'thonny.ui_u
ToplevelResponse	<class 'thonny.com
font	<module 'tkinter.fo
generate_event	<function generate

id: 0x363D278 << >>
repr: <module 'tkinter.font' from 'C:\\P
type: <class 'module'>

Attributes

Name	Value
BOLD	'bold'
Font	<class 'tkinter.f
ITALIC	'italic'
NORMAL	'normal'
ROMAN	'roman'
families	<function fami
itertools	<module 'itertc
names	<function nam
nametofont	<function nam
tkinter	<module 'tkint

Figure 1. Thonny's graphical user interface.

Editors Notebook

The Editors Notebook handles the management all of the opened files. From the point of view of the user, the only visible elements of the Editors Notebook are the tabs at the upper part of the screen, clicking on which activates the associated Editor and which display the name of the associated file for quick visual identification.

Editor

The Editor text area is situated in the middle of the Thonny window and provides a visual interface for viewing and editing the contents of the file associated with the currently active Editors Notebook tab. Vertical and horizontal scrollbars are dynamically enabled if the file contents do not fit inside the Editor area. Additional features assist with Python code generation, such as syntax coloring, parentheses matching and automatic indentation support.

Menu

The vertical Menu bar is located at the top of the Thonny window. Clicking on any header opens a list of Menu items, which have been grouped under headers based on their type and function. The contents of the menus are dynamic and some of the menu items may be disabled based on the current state information. For example, the **Save** item under the **File** menu is disabled if no Editors are currently opened.

Many of the menu items provide an alternative keyboard shortcut, for example the functionality of the **New** menu item under the **File** menu can also be invoked with the Ctrl+N key combination. The alternative keyboard shortcut is displayed after the menu item's name.

A few of the menu items provide access to functionality which can either be in enabled or disabled state. Clicking on the menu item or pressing the keyboard shortcut causes a switch to the opposite state. For example, the Variables view can either be hidden or shown. For such menu items, the current state of the corresponding functionality is indicated by having a check mark before the menu item's name when the functionality is enabled.

Shell

The Shell view, which is located directly underneath the Editor area, has two main functions: displaying the output of code runs and providing an interactive interpreter. Some basic output coloring is available, for example:

- Error messages are displayed in red, with the filename and line number of the incorrect syntax location displayed in blue.

- With every code run, the previous contents of the Shell are coloured gray, allowing the user to differentiate between the current and previous outputs.

File Browser

The File Browser component provides easy access to files overview and management. It can be activated and hidden by the user via the **View** menu. When activated, it is located to the left of the Editor area and the Shell view, and contains a tree view of nodes representing folders and files. Nodes containing further items can be expanded to list all of its sub-nodes or collapsed for a more compact view.

Views Pane

The Views pane is the container on the right side of the Editor area which contains the optional views that the user can enable and disable from the View menu. Prior to the development described in the current thesis, the following child views could be added to the Views pane:

- Variables view, which is used during debugging mode to show the table of variables and their values;
- Object inspector, which contains details about the object currently selected in the Variables view.

The Views pane is dynamically enabled and disabled as needed - if all of the components that are contained within the Views pane are disabled by the user, the Views pane is hidden and the vacated space automatically occupied by Editor area and Shell view instead. As soon as at least one of the views is enabled by the user, the Views pane is re-displayed and the Editor area and Shell view are dynamically resized to accommodate it.

Analysis of Thonny’s Usability

The analysis phase of the development project was carried out to identify some of the usability deficiencies of Thonny, based on which the scope of the project could then be established. First, the following chapter describes the methodology of the usability analysis along with the reasons why this approach was selected. The main findings of the usability analysis are then listed and finally the feature list scope of the development project is presented.

Methodology

Usability literature proposes many methodologies and perspectives for usability testing, including two approaches that can be broadly referred to as user testing and expert review [1, 2], both of which were considered for the development project. User testing involves having a representative group of users carry out a number of tasks using the analysed software and assessing their performance [1, 2]. Although it can be considered to provide more accurate results due to direct communication with representative end users, it also involves considerable effort. Expert review involves a usability review by a small number or even a single expert based on their subjective opinion or comparing the software program’s attributes to predetermined heuristic criteria [1, 10, 13]. This approach carries the risk of the decreased accuracy due to not considering the opinions of the representative end-users, but also has several advantages, for example the speed at which the analysis can be performed if the experts are readily available.

The preliminary and exploratory investigation of Thonny already revealed a number of usability deficiencies which were considered to be sufficiently significant to include in the list of concerns to be immediately addressed. As the aim of the usability analysis was not to provide a full and systematic evaluation of the usability of Thonny but rather to identify a small number of usability concerns which could realistically be improved by the subsequent development project, the decision was made that a sufficient project scope could be established based on a thorough expert review alone, considering the available project time. User testing was decided against as it seemed reasonable to assume that due to the nature of the project and the relatively young maturity level of Thonny, the effort required for creating the testing tasks, finding representative end users, carrying out the user observations and subsequently analysing user performance would not be an efficient use of the available project time. Furthermore, finding a representative user group would have been complicated as the main introductory programming course mandatory for first-year computer science students at the University of Tartu is held during the fall semester, but the analysis was performed at the beginning of the spring semester.

The usability research on Thonny was carried out by performing a variety of different tasks that were considered to be common for novice programmers and assessing the problems that users might encounter when carrying out these tasks. The aspects of usability that are important for novice programmers have been outlined previously in the current thesis. The analysis was performed by the author of the current thesis, based on his experience as a software developer, computer science student and programming teacher. Although any such analysis is admittedly subjective, the best attempt was made to consider the perspective of a novice programmer who has not used any other IDE before but is very likely familiar with common user interface solutions in widely used text editors and web browsers [7]. The fact that user testing was not conducted is an acknowledged risk of validity of the usability analysis and the current thesis as a whole, as is the author's lack of experience as a usability evaluator.

Results of the Analysis

The following subchapter presents some of the relevant results of the analysis. Generally, the usability of Thonny was found to be at a satisfactory level and in many ways equal to or better than that of IDLE. No performance or stability issues were identified and no functional software defects were discovered. However, certain usability areas were found to be in need of improvement. The most relevant findings of the analysis are listed below.

When a student first launches Thonny, they are shown just the Menu bar, an empty Editor and the Python shell. Other visual elements must be manually added from the View menu. Such initial lack of visual feature clutter seems to be a reasonable approach considering Thonny's end users, most of whom lack prior experience with IDEs and could thus be discouraged and confused by numerous interface elements that they do not comprehend. Instead, such approach allows gradually introducing new views and features as students gain more confidence in their abilities to use Thonny.

The fact that all visual components of Thonny are displayed in one window contrasts with IDLE, where each Python file is opened in a separate window instance and the Python shell is also in a different window. Although some users might prefer IDLE's solution, switching between windows can be inconvenient and stressful, especially if multiple files are open simultaneously. Having everything in one window allows the user to keep the focus on one visual space.

Thonny prevailingly uses what can be considered common graphical user interface design patterns which even novice programmers should already be familiar with based on their prior experiences with computers. For example, students have likely encountered user interfaces with a scrollable text area and an interactable menu bar when using Notepad or some other common text editor, while the concept of switching between tabs to switch contexts should be familiar from a modern web browser such as Chrome. The purpose of the Shell view might initially be unclear to students but should quickly become apparent during the initial lectures and practice sessions. Therefore, the fact that students can instantly utilize their previous knowledge of working with other content creation tools was considered a positive usability factor.

Providing a debugger accessible to novice programmers has been claimed by its creators to be one of the main reasons behind developing Thonny [17]. Visualization tools have been shown to positively contribute to students' understanding of complex algorithms and data structures [3, 12] and can therefore be considered a useful learning tool. Although the debugging mode interface seemed accessible in the expert opinion, the usability analysis of these features was considered inconclusive as correctly using these tools depends on the user's comprehension of the program execution flow [10, 15]. Therefore, conclusively evaluating the usability of Thonny's debugging tools seems to necessitate user testing on a representative group of end users. Thus, the decision was made to leave the possible usability improvements of Thonny's debugging tools out of the scope of the development project as more accurate analysis data would first be required.

The usability analysis revealed that Thonny did not provide sufficient assistance for convenient text navigation. There was also no functionality which would allow searching for a specific substring within the text, which can be considered a major usability deficiency as such a feature is available in nearly every text editor and IDE. Another feature that was unavailable in Thonny but seems to be common in other IDEs, including IDLE, is the ability to view a quick visual outline of the current program structure, such as its methods or classes declarations, and interacting with the elements of the outline to instantly move to the corresponding location within the source code. The lack of such features could make it needlessly difficult to write complex programs or to understand the implementation details of an unfamiliar Python module [13]. Thus, users had to spend time and energy on manually navigating the text or locating an occurrence of a substring, which could greatly distract from the programming process [13], and therefore the lack of these possibilities can be considered to have adversely affected the usability of Thonny.

Similarly, Thonny lacked tools which would assist the user with performing multiple related code changes, such as replacing one or more occurrences of a substring with another, or to intelligently rename an identifier so the change is propagated everywhere the identifier is referenced. Such features facilitate code management and refactoring which would otherwise have to be done manually, which can be tedious and time-consuming [13]. Other examples of features aimed to eliminate repetitive tasks that are often present in IDEs but were lacking in Thonny include assistance when typing identifiers and being able to add or remove comment characters at the beginning of multiple consecutive lines. Although such tools are not strictly necessary, they make it convenient to trigger code changes, thus increasing the users' comfort level with the IDE as well as allowing for more experimentation possibilities and encouraging creativity [13].

Final Project Scope

Based on Thonny's usability analysis and preliminary research into its technical implementation, the final project scope was established by Annamaa and the author of the current thesis. The scope consisted of a list of new features that were expected to improve Thonny's usability, especially in the areas of text modification and navigation. A variety of factors were taken into account when deciding on whether to include a possible improvement, such as its effect on usability, predicted frequency of use by students, the feature's presence in IDLE, and technical implementation difficulty. The best effort was made to put together the scope so that its completion would significantly improve some aspects of Thonny’s usability while realistically fitting within the project’s time schedule.

The finalized project scope consisted of the following items:

1. Find & Replace window
2. Autocomplete functionality
3. Outline view
4. Identifier name refactoring support
5. Block comment toggle support

In the next chapter, all of these features are described in more detail in their respective subchapter.

Development and Deliverables

The following chapter gives an overview of the deliverables of the development project, including the design considerations and implementation details of each of the added features. The entirety of the development was performed by the author of the current thesis. Aivar Annamaa as the owner and main developer of Thonny approved the design decisions and provided development advice. The source code modifications performed during the development phase can be viewed from the BitBucket commit history interface [20].

Development Principles

Before beginning development, a list of development principles was put together to act as guiding priorities when making design and implementation decisions. Strong effort was made throughout the design, development and testing phases to adhere to these principles in order to ensure the quality of the developed product:

1. **Stability**: under no circumstances can using the new features cause Thonny to irrecoverably crash or users to lose their unsaved work. To ensure this, Thonny was extensively tested on all supported operating systems after the development was completed and any found defects were fixed. This was followed by a full regression test after the completion of the implementation phase which passed without any issues.
2. **Modularity**: the new features must not be tightly coupled to each other and to the existing Thonny platform in their implementation. This allows future improvements of the features to be performed in isolation with a low risk of defects or stability issues in other modules, or for an implementation to be swapped out completely for a more preferable one. This proved to be difficult due to the large number of platform code modifications that were required. It was finally accomplished to a satisfactory extent as each of the new features was implemented in a separate Python module, with relatively loose coupling to the main Thonny platform. Furthermore, it is possible for each of the new features to completely disable them from Thonny configuration files on an individual basis.
3. **Integration**: the new features must look and feel as natural parts of the Thonny program to end users as well as to future Thonny contributors. This was considered to be successfully

accomplished for all aspects of the added features, from graphical implementation to coding style, all of which follow the standards established by the previously existing code.

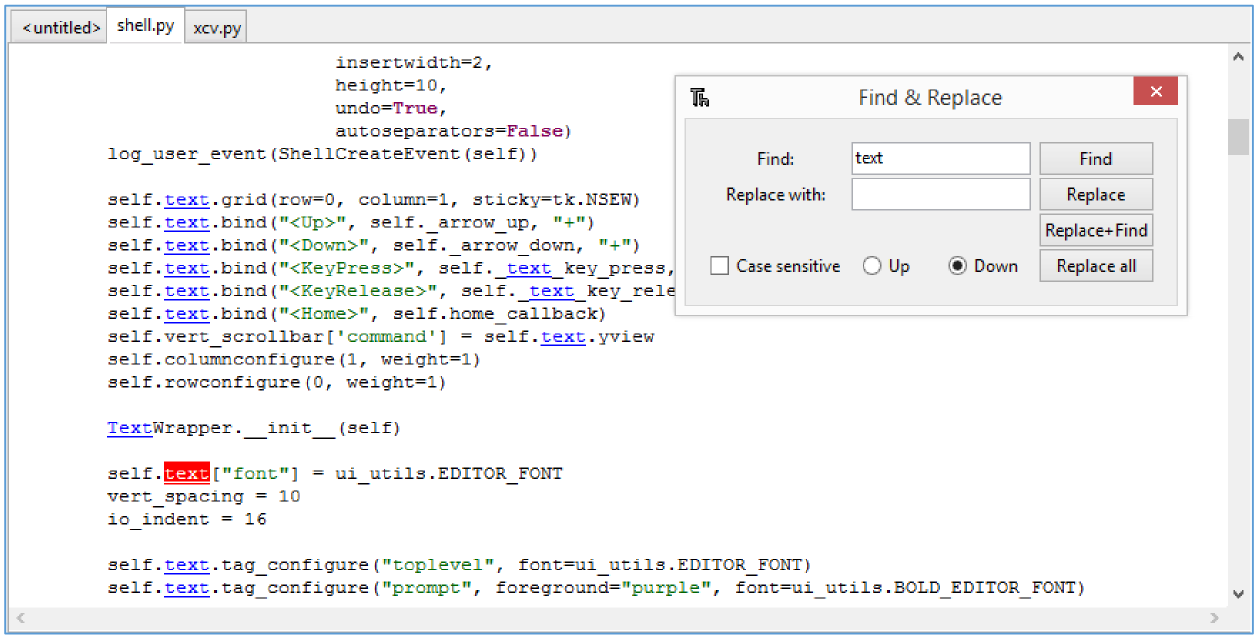
4. **Documentation**: to facilitate future code modifications by other developers, sufficient documentation of the delivered code would be needed, from relevant code comments to usage guides if needed. The choice was made to create a list of test suites to provide both requirements information as well as testing assistance. These test suites can be found in Appendix A of this thesis.
5. **Installability**: after the completion of the development project, installing and launching Thonny must be as simple as it was previously. This was accomplished fully with no additional setup steps introduced. All included third party libraries are bundled by default with the Thonny release version.

Find & Replace Window

The Find & Replace window is a modular pop-up window that is displayed on top of the Editor area and which allows users to search for specific text strings inside the contents of the currently active Editor, and to replace one or more of the occurrences of the found string with another string.

Implementation Overview

The Find & Replace window was implemented as a single top-level window containing various interactable text fields and buttons. While the window is active, all other Thonny elements are disabled, except for the Editor scrollbars which can be used to scroll the Editor's contents up and down to allow user navigation. The Find & Replace window can be activated by selecting the **Find & Replace** menu item from the **Edit** menu, or by pressing Ctrl+F key combination. The window can be hidden by clicking the close icon in the upper right corner of the pop-up window or by pressing the Esc key.



```
                            insertwidth=2,
                            height=10,
                            undo=True,
                            autoseparators=False)
        log_user_event(ShellCreateEvent(self))

        self.text.grid(row=0, column=1, sticky=tk.NSEW)
        self.text.bind("<Up>", self._arrow_up, "+")
        self.text.bind("<Down>", self._arrow_down, "+")
        self.text.bind("<KeyPress>", self._text_key_press,
        self.text.bind("<KeyRelease>", self._text_key_rele
        self.text.bind("<Home>", self.home_callback)
        self.vert_scrollbar['command'] = self.text.yview
        self.columnconfigure(1, weight=1)
        self.rowconfigure(0, weight=1)

        TextWrapper.__init__(self)

        self.text["font"] = ui_utils.EDITOR_FONT
        vert_spacing = 10
        io_indent = 16

        self.text.tag_configure("toplevel", font=ui_utils.EDITOR_FONT)
        self.text.tag_configure("prompt", foreground="purple", font=ui_utils.BOLD_EDITOR_FONT)
```

Find: text
Replace with:
☐ Case sensitive
● Down

Figure 2. Find & Replace window. Occurrences of the searched string “text” are highlighted.

When user triggers the searching action, the search is begun from the current position of the text cursor, or from the position of the last found occurrence in the case of a repeating search. If the next found occurrence is on a line that is not currently visible to the user, the Editor area is automatically scrolled to such a position that the line containing the currently active occurrence is visible. Furthermore, the last found occurrence is highlighted to the user by having a clearly distinguishable text style, as can be seen on the above screenshot. All other occurrences of the searched string other than the active selection are highlighted using a different foreground colour. If the string that the user searched for is not present in the current Editor’s contents, error text will be displayed to inform the user.

Each of the buttons performs a different action, chosen to mirror the functionality of the analogous find window of IDLE. The individual buttons are dynamically enabled and disabled based on the current state data. For example, if no text has been entered on the **Find** text field, all of the buttons are disabled, which is signaled to the user by graying out the button text and not responding to clicks.

Predicted Usability Improvements

The possibility of locating a string inside the Editor's contents should greatly improve the users' ability to quickly navigate code, especially in the case of large files. This can be expected to be especially useful if the code is not generated by the user and thus the user is not familiar with code structure, for example in the case of group projects. The ability to perform quick replacements of a specific string should facilitate performing code refactoring, for example replacing a inadequate variable name with a more informative one, while also helping to reduce the number of oversights and typing mistakes which might occur if these operations were performed manually.

It should also be noted that in some form, the functionality to find a specific substring in text or replacing it with another string seems to exist in very many if not nearly all widely used text editors, instant messaging programs and web browsers. It is reasonable to expect that most students are familiar with a common text editor such as Notepad or a web browser such as Chrome by the time they enter the introductory programming course, and have used the functionality of locating a piece of text, usually available via the Ctrl+F shortcut. The option to use this familiar feature is very likely to provide a sense of familiarity for a novice user and make them feel more comfortable when using Thonny.

Suggested Future Work

It could increase the amount of relevant information received by the user if the total number of occurrences when performing a search was also displayed. Currently, an error text appears within the Find & Replace window if the string that is searched for cannot be found. If the searched string, however, is found, the same window area could instead be used to display the total number of occurrences and the current occurrence's number. For example, if the user is currently at the 20th occurrence of the searched string and the Editor's text contains 34 occurrences of the string, the displayed text could be something similar to "20 of 34".

It might also improve the Find & Replace window's usability if additional search options were available. For example, many text editors provide an option of matching only the whole word, which usually means that the searched string is matched only when it is not surrounded by word characters on either side. Another useful feature could be the possibility to also search for a regex expression instead of just a string literal. These additional options should be analyzed and the ones found useful added, while keeping in mind that having too many options and interface elements could be confusing to the intended end user of Thonny.

Finally, some users might find it useful if there was a configurable setting to remember all of the search settings when the Find & Replace window is re-opened, rather than just the contents of the **Find** text field.

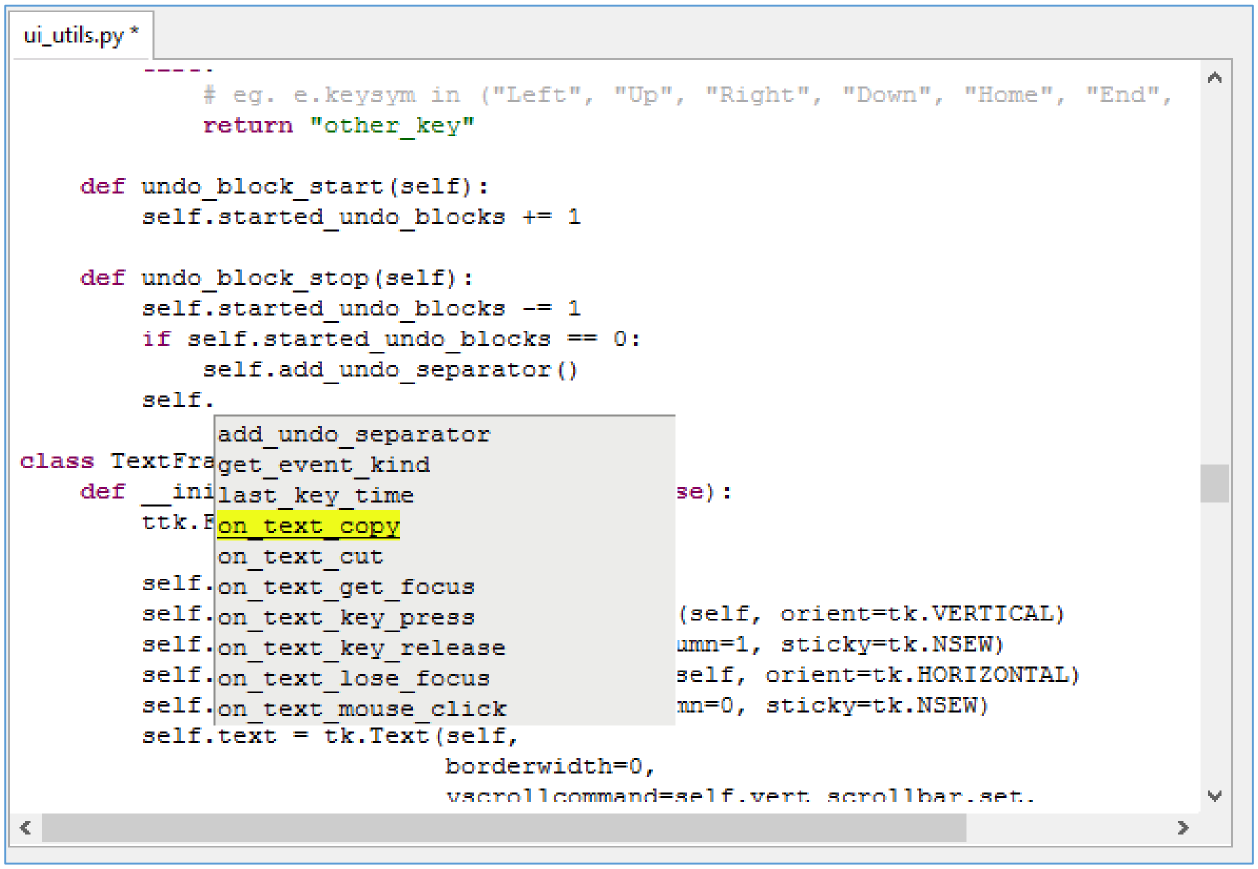
Autocomplete Functionality

The Autocomplete functionality allows quickly triggering a keyword or identifier completion based on the inserted partial string.

Implementation Overview

The decision was made to utilize the autocompletion functionality of the Jedi library [21]. Based on the initial research and testing it became apparent that Jedi would be relatively simple to integrate and seems to provide the all of the required backend functionality without adversely affecting the performance of Thonny. Therefore, given the complexity of independently creating an autocomplete library which is able to create the accurate list of suggestions based on the current code context and is able to parse Python files with incorrect syntax, integrating Jedi was deemed the most efficient choice.

From the user's point of view, autocompletion is triggered by placing the text cursor at the end of the partial string and choosing the **Autocomplete** menu item from the **Edit** menu or pressing Ctrl+Space. It was decided during the design phase that autocomplete feature is triggered only when the user explicitly requests it, rather than having an automatic popup which is present in some other IDEs but can be confusing for novice users.



```
        ----.
            # eg. e.keysym in ("Left", "Up", "Right", "Down", "Home", "End",
            return "other_key"

    def undo_block_start(self):
        self.started_undo_blocks += 1

    def undo_block_stop(self):
        self.started_undo_blocks -= 1
        if self.started_undo_blocks == 0:
            self.add_undo_separator()
        self.
             add_undo_separator
class TextFraget_event_kind
    def __inilast_key_time                          se):
        ttk.Fon_text_copy
             on_text_cut
        self.on_text_get_focus
        self.on_text_key_press                      (self, orient=tk.VERTICAL)
        self.on_text_key_release                    umn=1, sticky=tk.NSEW)
        self.on_text_lose_focus                     self, orient=tk.HORIZONTAL)
        self.on_text_mouse_click                    mn=0, sticky=tk.NSEW)
        self.text = tk.Text(self,
                              borderwidth=0,
                              vscrollcommand=self.vert_scrollbar.set,
```

Figure 3. Autocomplete suggestions box containing possible completions of “self.”.

The result of triggering the autocompletion functionality depends on the number of possible suggestions found by Jedi:

- If no suggestions matching the current partial string are found, no actions will take place.
- If only one matching suggestion is found, it is inserted automatically.
- If multiple suggestions are found, a list of them is displayed in the Autocompletion suggestions overlay, as can be seen on the screenshot above. The overlay will be situated so that the suggestions are directly aligned below the partial string. The displayed list of suggestions is limited to 10, but user can scroll through the list using the arrow keys or the mouse wheel. Selection is made by pressing the Enter key or double-clicking on a suggestion. The user can also exit without choosing any suggestions by pressing the Esc key or clicking outside the overlay area.

Predicted Usability Improvement

Autocompletion provides a quick and convenient mechanism for eliminating the repetitive and error-prone task of typing out the full name of long identifiers. Thus it should allow the user to keep their concentration on higher-level abstraction considerations instead, especially in the cases where the exact spelling of an identifier must be looked up from an external source. Having a convenient autocompletion feature might also help to eliminate a habit acquired by some students of using short non-descriptive variable names such as “a” and “b”, which is very likely caused by such names being faster to type.

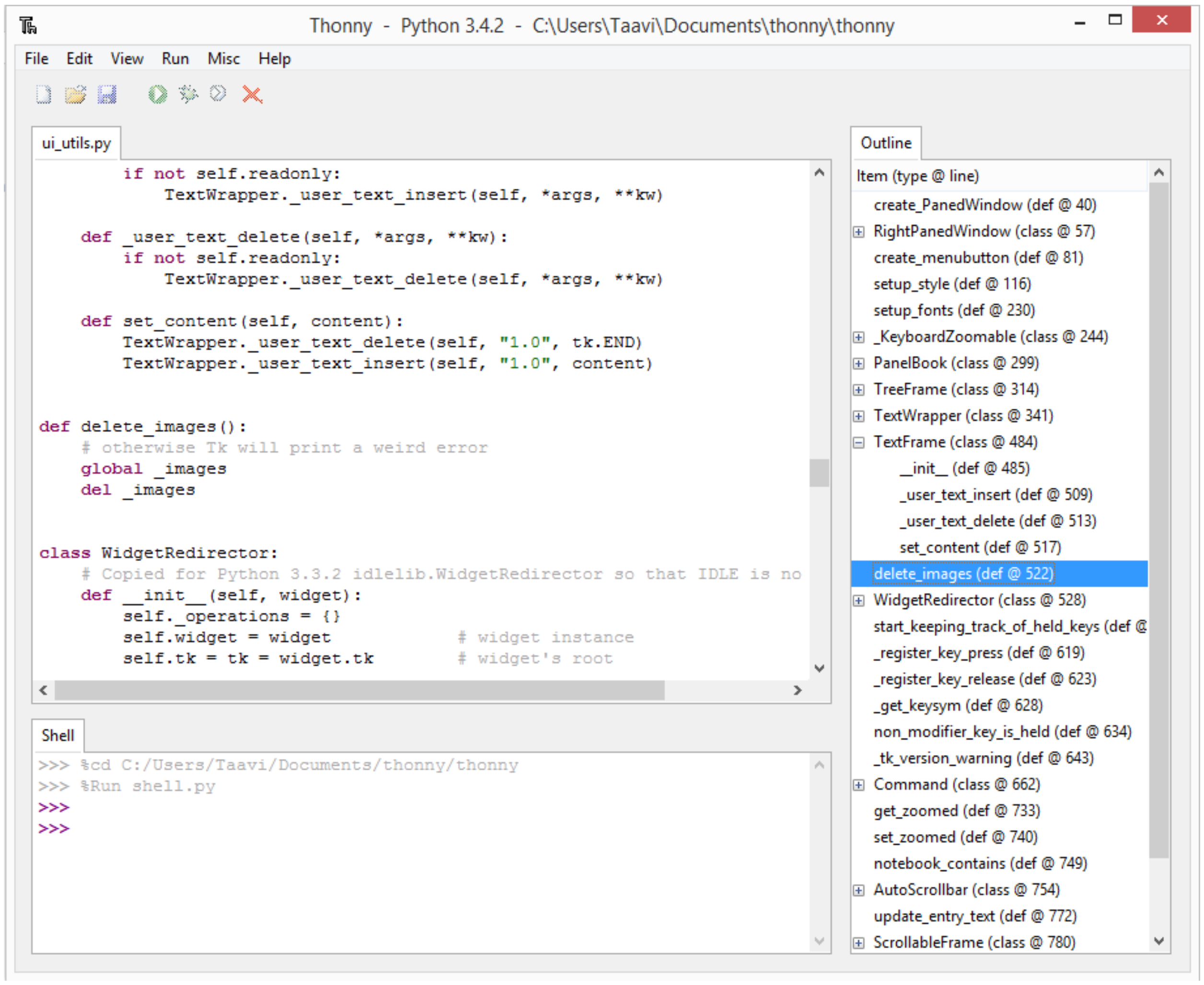
Another possible benefit of the autocompletion feature is that by frequently using the autocomplete functionality, the students become familiar with the list of members in Python standard library modules, which could encourage their creativity as they would then seek more information regarding methods with names that look useful for them for the task at hand. Furthermore, frequent exposure to the list of identifiers in Python’s standard libraries might help them to acquire certain common identifier naming practices or to understand the importance of using precise and descriptive identifiers.

Outline View

Outline view is a user interface element contained within the Views pane that provides a visual overview of the currently active source code’s structure by listing all the syntactically correct method and class definitions along with the corresponding line number. Each item can contain sub-items, for example a class node may also list methods declared within that class. Double-clicking on any node instantly moves the Editor’s viewport position so that the corresponding line becomes visible.

Implementation Overview

Outline view is implemented as a child element of the Views pane that can be activated and deactivated by selecting the **Show outline** menu item from the **View** menu. It is not possible to activate the Outline view if no Editors are currently open as the menu item itself will be disabled.


Thonny - Python 3.4.2 - C:\Users\Taavi\Documents\thonny\thonny
File Edit View Run Misc Help
ui_utils.py
Shell
Outline

```
        if not self.readonly:
            TextWrapper._user_text_insert(self, *args, **kw)

    def _user_text_delete(self, *args, **kw):
        if not self.readonly:
            TextWrapper._user_text_delete(self, *args, **kw)

    def set_content(self, content):
        TextWrapper._user_text_delete(self, "1.0", tk.END)
        TextWrapper._user_text_insert(self, "1.0", content)


def delete_images():
    # otherwise Tk will print a weird error
    global _images
    del _images


class WidgetRedirector:
    # Copied for Python 3.3.2 idlelib.WidgetRedirector so that IDLE is no
    def __init__(self, widget):
        self._operations = {}
        self.widget = widget            # widget instance
        self.tk = tk = widget.tk        # widget's root
```

```
>>> %cd C:/Users/Taavi/Documents/thonny/thonny
>>> %Run shell.py
>>>
>>>
```

```
Item (type @ line)
  create_PanedWindow (def @ 40)
⊞ RightPanedWindow (class @ 57)
  create_menubutton (def @ 81)
  setup_style (def @ 116)
  setup_fonts (def @ 230)
⊞ _KeyboardZoomable (class @ 244)
⊞ PanelBook (class @ 299)
⊞ TreeFrame (class @ 314)
⊞ TextWrapper (class @ 341)
⊟ TextFrame (class @ 484)
      __init__ (def @ 485)
      _user_text_insert (def @ 509)
      _user_text_delete (def @ 513)
      set_content (def @ 517)
  delete_images (def @ 522)
⊞ WidgetRedirector (class @ 528)
  start_keeping_track_of_held_keys (def @
  _register_key_press (def @ 619)
  _register_key_release (def @ 623)
  _get_keysym (def @ 628)
  non_modifier_key_is_held (def @ 634)
  _tk_version_warning (def @ 643)
⊞ Command (class @ 662)
  get_zoomed (def @ 733)
  set_zoomed (def @ 740)
  notebook_contains (def @ 749)
⊞ AutoScrollbar (class @ 754)
  update_entry_text (def @ 772)
⊞ ScrollableFrame (class @ 780)
```

Figure 4. Outline view containing the class and method structure of the "ui_utils.py" module.

When the Outline view is activated, it registers as a listener to both the currently active Editor as well as to the Editors Notebook. The current Editor's contents are then parsed to create a representation of its class and methods structure. It was decided that currently it is sufficient to use a simple regex which parses the Python module line by line, extracts all class and method nodes and places them in the correct position in the structure tree based on the indentation level of the line. Each time the Editor's contents change, it notifies all active listeners, including the Outline view, which then re-parses the entire file. The following regex is used to extract data from lines containing class and function declarations:

```
[  ]*(def|class){1}[  ]+[\w]+
```

Although such solution is quick and has the advantage of being able to parse syntactically incorrect Python files as no actual abstract syntax tree is created, it admittedly has some severe shortcomings. For example, in the case of the following lines, node for the function “test” would be added to the Outline view, although it is not an actual function definition but a syntactically valid string literal in Python:

```
"""
def test():
"""
```

These risks were analyzed and currently found to be acceptable.

The Editors Notebook tab change events are also listened to. Upon receiving a notification that the active tab has been switched, it unsubscribes from the previously active Editor and subscribes to the new one, then immediately parses the active Editor’s contents and repopulates the display. When the Outline view is deactivated, it unregisters itself from both the active Editor and the Editors Notebook.

Predicted Usability Improvement

As the Outline view contains the list of all classes and methods in the currently active file, users are able to conveniently look up an identifier, which could otherwise be a time-consuming task, especially in the case of larger files. Furthermore, this feature improves usability by reducing the time that the user has to manually navigate in code, as users can now just double-click on a node name to move to a specific location. Outline view can also be helpful when working with unfamiliar modules as it provides a concise introductory overview of the program structure before the implementation is examined in more detail [14].

Suggested Future Work

The main concern with the current implementation is that fully parsing the Editor’s contents after each code modification creates considerable and unnecessary overhead. Improving this logic could not be accomplished within the available timeline, but a future priority should be improving the implementation of the parsing logic so that only the modified lines are reparsed and the display is only updated when needed.

The appearance of the Outline view could also be improved so that more information is stored and displayed for each node. For example, method nodes could also contain information about the method's argument list, which would allow the user to look up a method's signature and order of arguments without having to navigate to the declaration. As horizontal space on the Views pane is limited, there would be no room for any new columns and therefore it would probably be reasonable to show this information as a tooltip when the user holds the mouse cursor over a node.

Identifier Refactoring Support

Identifier refactoring support is a feature that allows renaming Python identifiers so that the renaming action is intelligently propagated everywhere within the Python project where the identifier is references.

Implementation Overview

Based on the initial code analysis and research, it was estimated that the required effort for developing a module that provides full support for the backend logic of parsing all project files and resolving all identifier references is not viable within the available timeline. Therefore, the possibility of integrating a 3rd party library that provides this functionality was explored instead. After extensive research, the decision was made to integrate the Rope library [22]. As a complete refactoring library, Rope offers full code comprehension support and its API allowed convenient integration with Thonny.

From the user's point of view, identifier rename is triggered by positioning the text cursor within or adjacent to the identifier and selecting **Rename identifier** from the **Edit** menu. The user is then prompted to save all currently open files. Following this, a dialog window appears querying the user for a valid Python identifier until one is entered. The following regex is used to verify the name's validity and is based on information from Python's official reference [23]:

```
^[^\d\W]\w*\Z
```

After a valid new name has been entered, Rope internally performs the refactoring analysis and returns a preview list of change objects. At this point, the changes have not been performed yet.

Thonny parses this list to create a list of files that would be affected by the change, which is then displayed to the user. If the user confirms the changes, the renaming is performed by Rope. Thonny then reloads all the affected Editor tabs to ensure that their contents are up to date.

The user is offered several possibilities to cancel the renaming process. In addition, several error conditions are tested against throughout the process and if any errors occur, the whole flow is exited from as it is vitally important to ensure that either the renaming process is completed fully or it is cancelled without performing any changes.

Predicted Usability Improvement

Identifier rename is more likely to be used near the end of the introductory programming course when students are already working on more complex programs such as group projects. Due to Python’s dynamic name resolution system, users cannot always rely on compilation errors and would need to track down and manually change all references to an identifier in all of the files in a project if they decide to rename an identifier. This can be a stressful and time-consuming job, especially if several identifiers in possibly overlapping scopes share the same name. Using an intelligent refactoring engine to assist with this task can in some circumstances greatly improve usability by reducing the required time and effort. For example, it allows users to update identifiers as the program develops and the purpose of an identifier no longer corresponds to its initial name.

Suggested Future Work

A helpful feature that could be added is a way to undo the performed changes in all of the affected files immediately after performing the changes. To provide such an option, the relevant data must be available for Thonny. Possible solutions include creating a single file containing information or ‘diffs’ about all the actually performed changes, or creating a backup of all the affected files which could be restored.

Block Comment Toggle

Block comment toggle allows quickly commenting out a block of code by adding two comment symbols at the beginning of the source code lines, or uncommenting a block of code by removing up to two comment symbols from the beginning of the source code lines.

Implementation Overview

The user can add two comment symbols to the selected lines by clicking the **Comment in** item in the **Edit** menu or via the Ctrl+3 key combination. Up to two comment symbols can be removed via the **Comment out** item in the **Edit** menu or by pressing Ctrl+4. For both actions, the list of affected lines is first internally created. If no lines have been selected by the user, then only the line that currently contains the text cursor is added to the list. However, if the user has selected a number of lines, then all of the lines contained within the selected area will be affected.

Commenting out a block of code is a straightforward task of adding two comment characters to the beginning of all of the affected lines, regardless of how many comment characters are already present in the beginning of the line. Commenting in a block of code is somewhat more complex as the line might begin with only one comment character or even none at all, but only comment characters must be deleted. Therefore, first an algorithm determines how many comment characters the line begins with, and then two, one or none of them are deleted as necessary.

Predicted Usability Improvement

Commenting out a block of code seems to be an operation that many developers find themselves performing frequently, for example to provide an alternative implementation to a block of code but not erasing the previous one just in case. Having functions available in the IDE which help to do and undo this operation is convenient, quick and helps to avoid manual errors. Thus, the ability to toggle comments on a block of code seems to be present in most modern IDEs, including IDLE.

Future Thonny Development Suggestions

The following chapter lists some suggestions by the author of the current thesis to further improve Thonny usability or better it in some other way with future developments. The suggestions are based on the performed usability analysis as well as practical development experience with Thonny.

Currently one of the main obstacles to allowing any interested developers to contribute to Thonny seems to be that in order to add a new feature, the existing Thonny platform needs to be modified to add the necessary support. This was also the case during the development project described in the current thesis. As a result of the current design, all improvements must be partly implemented or at least extensively reviewed and tested by the main developers of Thonny to ensure program stability. The fact that platform and features are relatively tightly coupled also complicates simultaneous development by multiple developers and might create subtle defects after a new feature is added or when platform implementation itself is refactored. Thus, reviewing every new feature requires increasingly extensive effort from the main developers, which will likely not viable in the long term.

One of the possible solutions to this problem is providing a platform API which third party plugins can use to receive all the information they need and to send information about required changes to the platform. Designing and implementing such API is admittedly a complicated task, but provides the necessary isolation to allow anyone to create their own plugins for Thonny, which can then be simply disabled if they prove to contain defects. For example, coming up with new ideas for plugins and implementing them could be used as a possible way to receive extra credit in a programming course, with the plugins that have proved to be both useful and stable being added to Thonny’s standard releases, thus over time improving the program.

Another concern that should be addressed is the current lack of control over enabled features. As previously discussed, it seems reasonable to initially show students a simple and minimalist interface, with new features gradually enabled throughout the course. However, Thonny currently lacks a mechanism to conveniently provide this option to users. Implementing such mechanism requires an accessible interface where students could configure the current list of available features. For example, features could be organized into predetermined packages which users can switch between, with each successive level enabling additional and more advanced tools. Initially, students would use the “Beginner” setting, containing only the absolutely necessary features. When appropriate, they would switch to the “Advanced” feature package, which enables some additional tools, and so on. A separate option should be available to the more knowledgeable students to have full control over the list of enabled features as they wish.

In addition to the main concerns outlined above, various other end user features could improve Thonny's usability for novice programming students. Examples of such features or platform improvements include:

- Syntax correctness analysis, with the code segments containing incorrect syntax highlighted, underlined or otherwise marked;
- User help on Thonny’s features, such as a knowledge center or a help window which can be accessed online or from within the program. Another tutorial media that has been proven to be successful are tutorial videos [9] which, for example, could be created by students for extra course credit and uploaded to YouTube;
- Configurable user interface settings, e.g. syntax coloring rules or audio feedback;
- Displaying the line number values on the side of the Editor. Code examination revealed that this feature has actually been partially implemented but could not be enabled by users.

Finally, it should be mentioned that at the time of writing this thesis, plans are in place to provide the novice computer science students the opportunity to fully execute one or more of the test suites described in Appendix A of this thesis and report their findings for extra credit in a class. As a part of this exercise they are also encouraged to independently explore the new features and provide feedback regarding their usability and utility. Their feedback will be analyzed, any found defects fixed by the author of the current thesis and improvement suggestions forwarded to the main developers of Thonny.

Conclusion

The current thesis presented an overview of a software development project that was completed to improve the usability of Thonny, a Python IDE for novice programmers. As a preparatory step, a literature study in the fields of usability and computer science education was performed. Based on the results, a number of relevant quality components of the usability of a beginners' IDE were established. These criteria were then used as the basis of an expert review on the usability of Thonny, which revealed several shortcomings, most significantly in the areas of code navigation, program structure comprehension and batch change execution.

Some of the most significant Thonny's usability deficiencies were then addressed with the subsequent development project, which introduced a number of new platform features. An overview of each of the new features was presented in the thesis, including details on its implementation, information on the resulting usability improvement and suggestions for future enhancements. As a result of the development project, a number of Thonny's usability issues were corrected and Thonny should now be easier to use by novice programmers, hopefully leading to a more productive learning process.

Despite Thonny's generally high level of quality and a wide range of features which make it suitable for programming education, there is still considerable room for various improvements which would further increase its pedagogical value. A list of suggestions is presented as the last section of the thesis. The primary proposal is designing and implementing a platform API for extensions, which would allow conveniently integrating plugins developed by any interested party. Additionally, a novice-friendly interface should be implemented to allow users to configure the list of enabled features and plugins. Such approach should greatly contribute to Thonny's future developments and long term support.

References

[1] Nielsen, Jakob. "Usability 101: Introduction to Usability". 2012. Available at: http://www.nngroup.com/articles/usability-101-introduction-to-usability/ (last accessed 09.05.2015).

[2] Bevan, Nigel. "Usability." *Encyclopedia of Database Systems*. Springer US, 2009. 3247-3251. Available at: http://www.nigelbevan.com/papers/whatis92.pdf (last accessed 09.05.2015).

[3] Pears, Arnold, et al. "A survey of literature on the teaching of introductory programming." *ACM SIGCSE Bulletin* 39.4 (2007): 204-223. Available at: http://www.seas.upenn.edu/~eas285/Readings/Pears_SurveyTeachingIntroProgramming.pdf (last accessed 09.05.2015).

[4] Robins, Anthony, Janet Rountree, and Nathan Rountree. "Learning and teaching programming: A review and discussion." *Computer Science Education* 13.2 (2003): 137-172. Available at: http://130.216.33.163/courses/compsci747s2c/lectures/robins-learning-cse-03.pdf (last accessed 09.05.2015).

[5] Jenkins, Tony. "Teaching programming – A journey from teacher to motivator." *2nd Annual LTSN-ICS Conference*. 2001. Available at: http://www.cs.kent.ac.uk/people/staff/saf/dc/portfolios/tony/doc/other/motivation.pdf (last accessed 09.05.2015).

[6] Lahtinen, Essi, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. "A study of the difficulties of novice programmers." *ACM SIGCSE Bulletin*. Vol. 37. No. 3. ACM, 2005. Available at: https://student.brighton.ac.uk/mod_docs/cmis/past%20papers/ci_modules/level%20_2/2006_07/ci215_cs2_2006.pdf (last accessed 09.05.2015).

[7] Dillon, Edward, Monica Anderson-Herzog, and Marcus Brown. "Teaching Students to Program Using Visual Environments: Impetus for a Faulty Mental Model?" 2014. Available at: http://shodor.org/media/content/jocse/volume5/issue1/Dillon_2014.pdf (last accessed 09.05.2015).

[8] Holvikivi, Jaana. "Conditions for successful learning of programming skills." *Key Competencies in the Knowledge Society*. Springer Berlin Heidelberg, 2010. 155-164. Available at: https://hal.inria.fr/hal-01054703/document (last accessed 09.05.2015).

[9] Bennedsen, Jens, and Michael E. Caspersen. "Revealing the programming process." *ACM SIGCSE Bulletin*. Vol. 37. No. 1. ACM, 2005. Available at: http://users-cs.au.dk/mec/publications/conference/09--sigcse2005.pdf (last accessed 09.05.2015).

[10] Pane, John, and Brad Myers. "Usability issues in the design of novice programming systems." 1996. Available at: http://repository.cmu.edu/cgi/viewcontent.cgi?article=1818&context=isr (last accessed 09.05.2015).

[11] Ardito, Carmelo, et al. "Usability of e-learning tools." *Proceedings of the working conference on Advanced visual interfaces*. ACM, 2004. Available at: http://tesi.fabio.web.cs.unibo.it/twiki/pub/Tesi/DocumentiRitenutiUtili/p80-ardito.pdf (last accessed 09.05.2015).

[12] Dyke, Gregory. "Which aspects of novice programmers' usage of an IDE predict learning outcomes." *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.207.7498&rep=rep1&type=pdf (last accessed 09.05.2015).

[13] Squires, David, and Jenny Preece. "Predicting quality in educational software: Evaluating for learning, usability and the synergy between them." *Interacting with computers* 11.5 (1999): 467-483. Available at: http://www.irit.fr/recherches/ICS/projects/twintide/upload/446.pdf (last accessed 09.05.2015).

[14] Chen, Zhixiong, and Delia Marx. "Experiences with Eclipse IDE in programming courses." *Journal of Computing Sciences in Colleges* 21.2 (2005): 104-112. Available at: http://faculty.mercy.edu/facultyfiles/zchen1/papers/chenide.pdf (last accessed 09.05.2015).

[15] Reis, Charles, and Robert Cartwright. "Taming a professional IDE for the classroom." *ACM SIGCSE Bulletin*. Vol. 36. No. 1. ACM, 2004. Available at: http://www.drjava.org/papers/taming-ide-for-classroom.pdf (last accessed 09.05.2015).

[16] Budiu, Raluca. "Memory Recognition and Recall in User Interfaces". 2014. Available at: http://www.nngroup.com/articles/recognition-and-recall/ (last accessed 09.05.2015).

[17] Annamaa, Aivar. "Thonny, a Python IDE for Learning Programming." Unpublished. 2014.

[18] Python Software Foundation. “IDLE”. 2015. Available at: https://docs.python.org/3/library/idle.html (last accessed 09.05.2015).

[19] Thonny. Available at: https://bitbucket.org/plas/thonny (last accessed 09.05.2015).

[20] Thonny commits. Available at https://bitbucket.org/plas/thonny/commits/all (last accessed 09.05.2015).

[21] Jedi contributors. “Jedi - an awesome autocompletion/static analysis library for Python”. Available at: http://jedi.jedidjah.ch/en/latest/ (last accessed 09.05.2015).

[22] Rope contributors. “Rope, a python refactoring library ...“. Available at: http://rope.sourceforge.net/ (last accessed 09.05.2015).

[23] Python Software Foundation. “Lexical Analysis”. 2015. Available at: https://docs.python.org/3/reference/lexical_analysis.html (last accessed 09.05.2015).

Appendix A: Test Suites

The current thesis defines test suites as concise yet precise step-by-step instructions, with each step consisting of an action to be performed and its expected results. The expected program state before each step depends on the successful completion of the previous step and thus the steps must be performed exactly as described and sequentially in the listed order. Reproducing all of the listed steps and verifying the expected results covers all functional requirements and provides comprehensive test cases for regression testing. Failure of any of the individual test cases signals the failure of the test suite as a whole.

The test suites in this appendix are mainly intended for informational purposes for parties interested in the level of quality validation that was performed as part of the development project, as well as for future Thonny developers whose developments might affect the features added as part of this project. By following the described steps, any developers will be able to verify that their changes did not cause regression defects. Furthermore, the preciseness of the descriptions allows the test suites to be used as a basis for creating test scripts for automated testing, if this validation step is added to Thonny in the future.

The test suites will also be migrated to an online content sharing environment, facilitating future amendments and modifications as needed. Although the BitBucket environment does provide a wiki, the lack of certain markdown options makes it unsuited to host the test suites in their current format. At the time of writing the thesis, the decision for selecting the suitable environment has not yet been made.

Table of Contents

Guide to Test Suites 38
Glossary 39
Styles 39
Find & Replace Window 40
Glossary 40
Testing Steps 41
Autocomplete 46
Glossary 46
Testing Steps 47
Outline View 52
Glossary 52
Testing Steps 52
Identifier Refactoring Support 56
Glossary 56
Testing Steps 57
Block Comment Toggle 64
Testing Steps 64

Guide to Test Suites

In order to make the test suites more concise and easier to follow, certain terminology is consistently used throughout the test suites. This chapter will list and explain some of the terms that might not be instantly understandable to a reader. Additionally, this chapter will also describe the fonts and styles consistently used for denoting particular types of textual references.

Thonny's user interface elements are listed and explained in the chapter "Thonny's Graphical User Interface" of the thesis.

Glossary

Table 1 provides of some of the common terms used throughout the test suites. The introduction to each test suites chapter will also contain its own glossary for terms used only in that chapter.

Table 1. Test suites glossary.

Term	Description
text cursor	The flashing vertical line indicating where text will be entered.
check mark	The tick icon ✓ that precedes menu items when they are in the enabled state.
test file	Some of the test suites have the tester use a prepared Python file. Each test suite which uses one or more test files will provide a link where the file can be downloaded prior to beginning the test.
marked line	Test files contain significant lines which have been clearly marked with a unique identifier in the comments following the line.

Styles

Table 2 presents an overview of the text styles used throughout the test suites.

Table 2. Text suites styles.

Content type	Description	Example
Code listing	Courier New font on grey background.	`window = Window()`
Interface elements and paths	Black bolded font. Successive path elements are separated by the > character.	Click **Replace**. **File > Save**
URLs	Blue underlined font.	www.example.com
Key combinations	Green bolded font. If several keys need to be pressed simultaneously, the keys are separated by the + character.	**Alt+C**
Marked line	Red bolded font.	Navigate to the marked line **LINE 1**.

Find & Replace Window

The test suite will use one test file, which has been prepared and must be downloaded by the tester. During the testing, a variety of actions related to searching and replacing will be performed. Testing will also verify the correctness of the graphical layout.

Glossary

Table 3 lists the terms used throughout the test suite.

Table 3. Find & Replace window glossary.

Term	Description
enabled / disabled buttons	Buttons can be in either enabled or disabled state. When enabled, buttons can be clicked to perform an action and have black text. Disabled buttons cannot be clicked and have grey text.
focused	When performing a search operation, one occurrence of the searched string is focused in the Editor area. When repeating the search, focus will move to the next occurrence of the string. The focused string can be visually distinguished by using a font with red background and white foreground.
highlighted	When performing a search operation, all occurrences of the searched string (except for the focused occurrence) are highlighted in the Editor area. The highlighted strings can be visually distinguished by using a font with blue foreground.

Interface Elements

Table 4 describes the interface elements of the **Find & Replace** dialog window.

Table 4. Find & Replace window interface elements.

Interface Element	Description
Find button	Performs a new search for the string in the **Find** text field.

Replace button	Replaces the currently active found occurrence with the string in the **Replace with** text field.
Replace & Find button	Replaces the currently active found occurrence and performs a new search.
Replace All button	Replaces all found occurrences of the string in the **Find** text field with the string in the **Replace with** text field.
Up and **Down** radiobuttons	Mutually exclusive. Determine the search direction.
Find text field	Allows the user to enter the string to be searched for.
Replace with text field	Allows the user to enter the string to replace the found string. If the **Replace with** string is empty, the found string will be simply deleted.
Match case checkbox	If checked, only strings which are case sensitively equal to the string in the **Find** text field will be considered matched.

Testing Steps

Table 5 lists the testing steps in the order that they must be performed.

Table 5. Find & Replace window testing steps.

Step #	Actions	Expected result
1	Download the test file from: https://bitbucket.org/plas/thonny/raw/default/thonny/tests/find_replace_test.py Open it with Thonny. **File > Open… > Select file > Open**	The file contents open in the Editor.

2	Open the **Edit** menu and select **Find**.	The **Find & Replace** window appears. Verify that initially the window is located on top of the Editor area but can be moved around by holding down the left mouse button and dragging it from the title bar. Do not type anything yet, but verify that the text cursor is active in the **Find** text field. All of the buttons (**Find**, **Replace**, **Replace+Find** and **Replace All**) are disabled since there is nothing in the **Find** text field.
3	Enter any text in the **Find** text field.	Verify that **Find** and **Replace All** buttons become enabled.
4	Delete all of the text that you entered in the **Find** text field.	All of the buttons (**Find**, **Replace**, **Replace+Find** and **Replace All**) are again disabled since there is nothing in the **Find** text field.
5	Attempt to click on several locations within the main Thonny window but outside the **Find & Replace** window, including on the Thonny window's close (X) button, menu buttons, etc.	Verify that nothing happens - clicking on any of the Thonny elements except the **Find & Replace** window must be disabled.
6	Place the mouse cursor within the Editor area but outside the **Find & Replace** window. Using the mouse wheel, attempt to scroll the Editor contents.	Using the mouse wheel to scroll the Editor contents up or down must work as normally.
7	Click the close (X) button of the **Find & Replace** window.	The **Find & Replace** window closes. All of the elements of the Thonny main window are re-enabled.
8	Ensure that the text cursor is at the start (before the very first character) of the Editor contents. Press **Ctrl+F**.	The **Find & Replace** window appears.

9	Insert “time” in the **Find** text field. Ensure that the **Down** radio button is checked. Press **Enter** to perform the search.	The marked line **LINE 1** is visible in the Editor area. The text string `time` within the marked line is focused. Use the mouse wheel to scroll through the Editor area and verify that no other text strings are focused, and the only highlighted text string is `time` on the marked line **LINE 2**. Verify that the **Replace** and **Replace+Find** buttons are enabled.
10	Click **Find** to repeat the search.	The contents of the Editor area are automatically moved so that the marked line **LINE 2** is visible. The substring `time` within the marked line is focused. Use the mouse wheel to scroll through the Editor area contents and verify that no other text strings are focused, and the only highlighted substring is on the marked line **LINE 1**.
11	Click **Find** to repeat the search.	The contents of the Editor area are automatically moved so that the marked line **LINE 1** is visible and its substring `time` is focused.
12	Press **Esc** to close the **Find & Replace** window.	The **Find & Replace** window closes. Verify that the substring `time` on the marked lines **LINE 1** and **LINE 2** are no longer focused and highlighted, respectively.
13	Insert the text cursor at the start (before the very first character) of the Editor contents. Press **Ctrl+F** to open the **Find & Replace** window.	Verify that `time`, which was the last string searched for, is still on the **Find** text field. Also verify that the **Find** and **Replace All** buttons are enabled.

14	Delete the previous string from the **Find** text field and insert the string “base64”. Check the **Up** radio button. Check the **Case sensitivity** checkbox. Click **Find**.	The contents of the Editor area are automatically moved so that **LINE 3** is visible and that the text string sys is focused.
15	Click **Find** to repeat the search.	The contents of the Editor area are automatically moved so that **LINE 4** is visible and that the text string `base64` is focused.
16	Delete the previous string from the **Find** text field and insert the string “aa”. Click **Find**.	The text “The inserted string can’t be found” appears underneath the **Replace with** text field. The viewport of the Editor area is not moved. Use the mouse wheel to scroll through the Editor area contents and verify that no text string are focused or highlighted. The **Replace** and **Replace+Find** buttons must now be disabled.
17	Delete the previous string from the **Find** text field and insert the string “a”. Click **Find**.	The error text “The inserted string can’t be found” disappears from underneath the **Replace with** text field.
18	Delete the previous string from the **Find** text field and insert the string “banner”. Check the **Case sensitive** check box. Click **Find**.	The text “The inserted string can’t be found” again appears underneath the **Replace with** text field.
19	Close and then reopen the **Find & Replace** window.	The string “The inserted can’t be found” is not present underneath the **Replace with** text field.

20	Delete the previous string from the **Find** text field and insert the string “import sys”. Click **Find**.	The contents of the Editor area are automatically moved so that **LINE 5** is visible and its substring `import sys` is focused.
21	Insert the string “from sys import *” in the **Replace with** text field. Click **Replace**.	The substring `import sys` is removed from **LINE 5** and it now contains `from sys import *` instead. Use the mouse wheel to scroll through the Editor area contents and verify that no text string are focused or highlighted, including the modified line. The **Replace** and **Replace+Find** buttons must now be disabled.
22	Insert the string “bestmatch” in the **Find** text field. Click **Find**.	The text string `bestmatch` on the line **LINE 6** is focused.
23	Click **Replace & Find**.	Verify that the substring `bestmatch` is removed from **LINE 6** and it now contains `best_match` instead. The text string `bestmatch` on **LINE 7** is now focused.
24	Insert the string “best_match” in the **Replace with** text field. Click **Replace & Find**.	Verify that the substring `bestmatch` is removed from **LINE 7** and it now contains `best_match` instead. The text “The inserted string can’t be found” appears underneath the **Replace with** text field.
25	Insert the string “sys.” in the **Find** text field. Insert an empty string (nothing, not even any spaces) in the **Replace with** text field. Click **Replace All**.	Use the mouse wheel to scroll through the Editor area and verify that the substring `sys.` is not present on **LINE 8** and it now contains just `stdout.flush()`

Autocomplete

The test suite will use one test file, which has been prepared and must be downloaded by the tester. During the testing, different autocompletion actions will be attempted (e.g. no suggestions found for currently unfinished string, one suggestion found, several suggestions found, and so on). The correctness of the graphical layout will also be verified. It will also be verified that following into imports and other complex syntax comprehension tasks are done correctly.

Glossary

Table 6 lists the terms used throughout the test suite.

Table 6. Autocomplete glossary.

Term	Description
suggestion	A single suggested completion for the current autocomplete action.
suggestions list	The list of suggestions contained within the autocomplete box.
autocomplete box	A rectangle-shaped overlay appearing on top of the Editor area and containing the suggestions list. Only appears if more than one suggestions are available. See chapter "Autocomplete Functionality" of the thesis for a screenshot of the autocomplete box.
|	Text representation of the position of the text cursor. It is displayed in red bold font for better visibility.

Testing Steps

Table 7 lists the testing steps in the order that they must be performed.

Table 7. Autocomplete testing steps.

Step #	Actions	Expected result
1	Download the test file from: https://bitbucket.org/plas/thonny/raw/default/thonny/tests/autocomplete_test.py Open it with Thonny. **File > Open… > Select file > Open**	The file contents open in the Editor.
2	Find the marked line **LINE 1** and insert the text cursor at the end of the line. Make you sure leave a space before the preceding character and the text cursor: `import |` Select **Autocomplete** from the **Edit** menu.	The **Autocomplete** box opens under the current line, containing module names that can be imported. The module names are in alphabetic order. The very first module name is in focus, which can be determined by it being underlined and having a different background colour than the other options.
3	Position the mouse cursor within the Autocomplete box. Use the mouse wheel to scroll up and down.	The list of suggestions must intuitively change according to the wheel scroll direction in alphabetically ascending order. Verify that when you have scrolled to the top of the list, attempting to scroll even further up does not cause any errors. Similarly, attempt to scroll further down when you are at the bottom and verify that no errors occur. The focused suggestion must not change as you are scrolling, meaning that if you scroll further down where the focused name is no longer visible, none of the visible suggestions appear to be focused. As you scroll back up, the first suggestions must still be in focus.

		As you are scrolling through the list of suggestions, verify that they are in alphabetical order and that names starting with uppercase and lowercase letters are present.
4	Using the mouse wheel, scroll the suggestions list to such a position where the focused suggestion is not visible. Press the down arrow key.	The suggestions list is automatically moved to the position where the new focused suggestion is visible. The new focused item is one position below the previously focused one.
5	Press up and down arrow keys.	Verify that pressing the up arrow focuses the item directly above the previously focused one. Similarly, pressing the down arrow focuses the item directly below the previously focused one. When the new focused item is not visible, the suggestions list is automatically moved so that the new focused item is visible to the user. Verify that when the focused item is the very first item in the suggestions list, pressing the up arrow key does nothing and does not cause any errors. Similarly, pressing the down arrow key when the focused item is the very last item in the suggestions list must simply do nothing and not cause any errors.
6	Perform a single left-click on the focused item in the suggestions list.	Verify that nothing happens.
7	Perform a single left-click on an item in the suggestions list that is not the currently focused item.	The item that you clicked on must now become the focused item.
8	Left-click in the Editor area, outside of the Autocomplete box.	The Autocomplete box closes, allowing the user to continue using the Editor. The Editor contents are not modified as no suggestions were chosen by the user.

9	Find the marked line **LINE 2** and insert the text cursor at the end of the line: `import p|` Press **Ctrl + Spacebar**.	The Autocomplete box opens under the current line, containing module names that can be imported. All suggestions must begin with either lowercase or uppercase “p”. The position of the Autocomplete box must be such that the first letters of the suggestions are directly below the “p” character.
10	Press **Esc**.	The Autocomplete box closes, allowing the user to continue using the Editor. The Editor contents are not modified as no suggestions were chosen by the user.
11	Find the marked line **LINE 3** and insert the text cursor at the end of the line: `sys.|` Activate Autocomplete. NB! As both Autocomplete activation methods have been tested, starting with this step, Autocomplete can be activated either via the Menu or via the key combination, as preferred by the tester.	The Autocomplete box opens under the current line, containing members of the `sys` module that are available to the user. For reference, these should include `api_version`, `argv` and `base_exec_prefix`.
12	Click on the **File** menu item.	The Autocomplete box disappears. The Editor contents are not modified as no suggestions were chosen by the user. The **File** menu is opened as usually.
13	Find the marked line **LINE 4** and insert the text cursor at the end of the line: `getfilesys|` Activate Autocomplete.	Verify that nothing happens. No suggestions must be available to the user because they must access the `getfilesystemencoding` member of the `sys` package as `sys.getfilesystemencoding` due to the import syntax used.

14	Find **LINE 5** and insert the text cursor at the end of the line: `os.path.|` Activate Autocomplete.	Verify that nothing happens. No suggestions must be available to the user because they must access the `path` member of the `os` module as `path` and not as `os.path` due to the import syntax used.
15	Find **LINE 6** and insert the text cursor at the end of the line: `path.|` Activate Autocomplete.	The Autocomplete box opens under the current line, containing members of the `os.path` submodule that are available to the user.
16	Double-click on the `dirname` suggestion.	Autocomplete box disappears. The string `dirname` must be appended to the line so that it is now: `path.dirname`
17	Find **LINE 7** and insert the text cursor at the end of the line: `re.|` Activate Autocomplete.	Verify that nothing happens. No suggestions must be available to the user because they must access the members of the `re` module directly by their name and not via the `re` reference due to the import syntax used.
18	Find **LINE 8** and insert the text cursor at the end of the line: `compi|` Activate Autocomplete.	Autocomplete box is not displayed. Instead, the string `le` is added directly to the line as it was the only suggestion. The line must now be: `compile` Note that `compile` is a member of the `re` package but must be referenced directly, without the `re` module reference, due to the import syntax used.
19	Find **LINE 9** and insert the text cursor at the end of the line: `testob|` Activate Autocomplete.	Verify that nothing happens. No suggestions must be available to the user because even though `testobject` is correctly declared in this file, the variable declaration happens after the current line.

20	Find **LINE 10** and insert the cursor at the end of the line: `testo|` Activate Autocomplete.	Autocomplete box is not displayed. Instead, the string `bject` is added directly to the line as it was the only suggestion. The line must now be: `testobject`
21	Find **LINE 11** and insert the text cursor at the end of the line: `testobject.|` Activate Autocomplete.	The Autocomplete box opens under the current line, containing all the correctly defined members of the `TestClass` instance: `instanceMethod`, `staticMethod` and `variable`. Verify that the Autocomplete box is correctly sized and is not longer than it needs to be for containing the three items.

Outline View

The test suite will use one test file, which has been prepared and must be downloaded by the tester. Testing will verify that the Outline view display has the correct graphical layout and that it corresponds to the actual file structure. Tests also confirm that the Outline view is automatically updated as new items are added and previous ones removed.

Glossary

Table 8 lists the terms used throughout the test suite.

Table 8. Outline view glossary.

Term	Description
node	Outline view item representing the start of a class or method definition.
child node	An indented node representing a nested class or method definition.
expand	Nodes can be expanded by clicking on the adjacent + icon. When expanded, all the children of the node are displayed.
collapse	Nodes can be collapsed by clicking on the adjacent – icon. When collapsed, the children of the node are hidden.

Testing Steps

Table 9 lists the testing steps in the order that they must be performed.

Table 9. Outline view testing steps.

Step #	Actions	Expected result
1	Download the test file from: https://bitbucket.org/plas/thonny/raw/default/thonny/tests/outline_view_test.py Open it with Thonny. **File > Open… > Select file > Open**	The file contents open in the Editor.

2	If the **Views** pane contains any views, such as the **Variables** view or the **Object Info** view, disable them from the **View** menu. Active views can be recognized by the preceding check mark in the **View** menu.	**Views** pane disappears.
3	Enable the **Outline** view from the **View** menu. **View > Show outline**	**Views** pane appears, **Outline** view is the only item contained in it. Verify that the check mark is present in front of the **Show outline** item in **View** menu.
4	Enable **Variables** view.	**Views** pane stays visible and the **Variables** view is added to it. Both **Outline** and **Variables** views are now visible.
5	Disable the **Outline** view.	**Views** pane stays visible and the **Outline** view is removed from it. Only **Variables** view is now visible. Verify that the check mark is not present in front of the **Show outline** item in **View** menu.
6	Enable the **Outline** view.	**Views** pane stays visible and the **Outline** view is added to it. Both **Outline** and **Variables** views are now visible.
7	Disable **Variables** view.	**Views** pane stays visible and the **Variables** view is removed from it. Only **Outline** view is now visible.
8	Verify that the **Outline** view contents correspond to the file structure. Do not yet expand any of the nodes.	Expected structure: **+ AdditionClass** **+ SubtractionClass** Both of these nodes have the expand icon next to them.

9	Expand the **AdditionClass** node.	The **AdditionClass** node expands, revealing the **perform_addition** child node, which itself can be expanded. The expand icon next to **AdditionClass** node changes to the collapse icon. The expected structure at this point: **- AdditionClass** **+ perform_addition** **+ SubtractionClass**
10	Expand the **perform_addition** node.	The **perform_addition** node expands, revealing the **add** child node, which cannot be expanded. The expand icon next to **perform_addition** node changes to a collapse icon. The expected structure at this point: **- AdditionClass** **- perform_addition** **add** **+ SubtractionClass**
11	Double-click on the **SubtractionClass** node.	The Editor is automatically scrolled to a position where the user can see the line `class SubtractionClass:`
12	Double-click on the **AdditionClass** node.	The Editor is automatically scrolled to a position where the user can see the line `class AdditionClass:`
13	Double-click on the header line **Item (type @ line)**	Nothing happens. Note: this step is needed for regression testing as it used to cause an error message popup.
14	Find the line `class AdditionClass:` in the Editor. Change it by replacing `AdditionClass` with `Test`. The line must now be: `class Test:`	In the **Outline** view, the **AdditionClass** node name is changed to **Test**.

15	Find the line `def subtract(x, y):` in the Editor. Change it by replacing `def` with `deff`. Line must now be: `deff subtract(x, y)`	Expand the **SubtractionClass** and **perform_subtraction** nodes. Verify that the **subtract** node is not present in the **Outline** view.
16	Find the line `def add(x, y):` in the Editor. Comment out the line by inserting the `#` symbol as the first character. Line must now be: `#def add(x, y):`	Expand the **AdditionClass** and **perform_addition** nodes. Verify that the **add** node is not present.
17	Comment in the same line by removing the `#` symbol. Line must now be: `def add(x, y):`	Expand the **AdditionClas**s and **perform_addition** nodes. Verify that the **add** node is re-added to its correct location as a child of **perform_addition** node.
18	Disable the **Outline** view.	**Views** pane disappears. Verify that the check mark is not present in front of the **Show outline** item in **View** menu.

Identifier Refactoring Support

The test suite will use two test files, which have been prepared and must be downloaded by the tester. One of these files, called the target file, will import various variables, methods and classes from the other file, called the source file. During the testing, the imported identifiers will be renamed. Then it will be verified that the renaming is correctly performed in both files. All known and foreseen error cases and edge cases will also be tested.

Glossary

Table 10 lists the terms used throughout the test suite.

Table 10. Identifier refactoring support glossary.

Term	Description
test folder	The folder containing the two test files used in this test suite.
source file / source file tab	This file contains the various identifiers that are imported by the target file.
target file / target file tab	This file imports various identifiers from the source file.

Testing Steps

Table 11 lists the testing steps in the order that they must be performed.

Table 11. Identifier refactoring support testing steps.

Step #	Actions	Expected result
1	Create a new folder anywhere on the hard drive which will be used to contain nothing but the test files. Download the test files from: https://bitbucket.org/plas/thonny/raw/default/thonny/tests/source_refactor_rename_test.py https://bitbucket.org/plas/thonny/raw/default/thonny/tests/target_refactor_rename_test.py NB! Save them both in the test folder. Open both of the downloaded files with Thonny. **File > Open… > Select file > Open** If there are any other open tabs except for those containing the test files, close them.	Thonny contains exactly two tabs, each containing one of the test files.
2	Activate the source file tab and insert a new empty line somewhere in the file contents.	The tab name must now be followed by an asterisk, indicating that file contents have been modified since the file was last saved: **source_refactor_rename_test.py ***
3	On line marked as **LINE S1**, place the text cursor anywhere within the `Calculator` identifier. Select **Rename identifier** from the **Edit** menu.	**Save Files Before Rename** window appears, requesting user to confirm that they want to save files before they can proceed. The window must contain two buttons: **Yes** and **No**.

4	Click **No**.	The refactoring process will be canceled: The **Save Files Before Rename** window must disappear and no further dialog windows or errors will be displayed. All other Thonny elements will be enabled again, allowing the user to continue using the program.
5	Open a new Editors Notebook tab. **File > New**	A new tab is created, with **<untitled>** as its name.
6	Activate the source file tab. On line **LINE S1**, place the text cursor within the `Calculator` identifier. Select **Rename identifier** from the **Edit** menu. Click **Yes** to save modified files.	**Save As** window appears, requesting user to choose the location and filename of the new Editors Notebook tab's contents. The **<untitled>** tab will be activated - this allows the user to see which tab they are saving in the case there are more than one new tabs.
7	Click **Cancel** to cancel saving the **<untitled>** tab to a file.	The refactoring process will be canceled: The **Save As** window must disappear. The **Rename failed** error popup must appear, informing the user that there are unsaved tabs and refactoring process cannot continue. The only available button is **OK**.
8	Click **OK**.	The **Rename failed** error popup must disappear and no further dialog windows or errors will be displayed. All other Thonny elements will be enabled again, allowing the user to continue using the program.

9	Activate the source file tab. On line **LINE S1**, place the text cursor within the `Calculator` identifier. Select **Rename identifier** from the **Edit** menu. Click **Yes** to save modified files. In the **Save As** window, navigate to the test folder and select it, so the file would be saved there. Enter "newfile_refactor_rename_test.py" as the filename and click **Save**.	Verify that the test folder now contains an additional file named "newfile_refactor_rename_test.py". The tab name of the previously untitled tab must now be **newfile_refactor_rename_test.py**. The **Rename** window appears, requesting user to choose the new name for the identifier. It must contain the **New name** text field as well as **OK** and **Cancel** buttons.
10	Click **Cancel**.	The refactoring process will be canceled: The **Rename** window must disappear and no further dialog windows or errors will be displayed. All other Thonny elements will be enabled again, allowing the user to continue using the program.
11	Activate the source file tab. On line **LINE S1**, place the text cursor within the `class` keyword. Select **Rename identifier** from the **Edit** menu.	Since at this point none of the tabs contain unsaved changes, showing the **Save Files Before Rename** window must be skipped and the **Rename** window is displayed instead.

12	Enter “newname” in the **New name** text field and click **OK.**	The refactoring process will be canceled because the text cursor was not placed within an identifier. The **Rename** window must disappear. The **Rename failed** error popup must appear, informing the user that an error was encountered and listing a few possible reasons. The only available button is **OK**.
13	Click **OK**.	The **Rename failed** error popup must disappear and no further dialog windows or errors will be displayed. All other Thonny elements will be enabled again, allowing the user to continue using the program.
14	Activate the source file tab. On line **LINE S2**, place the text cursor within the `sumtogether` identifier. Select **Rename identifier** from the **Edit** menu. In the **Rename** window, enter “sumtogether” in the **New name** text field and click **OK**.	The refactoring process will be canceled because the new identifier entered by the user is exactly the same as the existing name. The **Rename** window must disappear. The **Rename failed** error popup must appear, informing the user that no identifiers would be affected by the change. The only available button is **OK**.
15	Click **OK**.	The **Rename failed** error popup must disappear and no further dialog windows or errors will be displayed. All other Thonny elements will be enabled again, allowing the user to continue using the program.

16	On line LINE2, place the text cursor within the `sumtogether` identifier. Select **Rename identifier** from the **Edit** menu. In the **Rename** window, enter “1sum1” in the **New name** text field and click **OK**.	User must re-enter the identifier because the one entered now is not a valid Python identifier. The **Rename** window must disappear. The **Incorrent identifier** error popup must appear, informing the user that the name was not a correct Python identifier.
17	Click **OK**.	The **Incorrent identifier** error popup must disappear and the **Rename** window appear again, allowing the user to enter a valid name.
18	In the **Rename** window, enter " " (a single space) in the **New name** text field and click **OK**.	User must re-enter the identifier because the one entered now is not a valid Python identifier. The **Rename** window must disappear. The **Incorrent identifier** error popup must appear, informing the user that the name was not a correct Python identifier.
19	Click **OK** to close the **Incorrect identifier** popup. In the **Rename** window, enter “(add)” in the **New name** text field and click **OK**.	User must re-enter the identifier because the one entered now is not a valid Python identifier. The **Rename** window must disappear. The **Incorrent identifier** error popup must appear, informing the user that the name was not a correct Python identifier.
20	Click **OK** to close the **Incorrect identifier** popup. In the **Rename** window, enter “add” in the **New name** text field and click **OK**.	The entered name is a valid Python identifier so the refactoring flow must proceed normally: The **Confirm changes** dialog window must appear. It must inform user that the following files will be modified during refactoring: ▪ source_refactor_rename_test.py ▪ target_refactor_rename_test.py The window must contain the **Yes** and **No** buttons, allowing user to proceed with refactoring or to cancel it, respectively.

21	Click **No**.	The refactoring process will be canceled: The **Confirm changes** window must disappear and no further dialog windows or errors will be displayed. All other Thonny elements will be enabled again, allowing the user to continue using the program.
22	Activate the source tab. Enable the **Outline** view, if it was not enabled already: **View > Show outline**	**Views** pane is visible and now contains the **Outline** view.
23	On line **LINE S2**, place the text cursor within the `sumtogether` identifier. Select **Rename identifier** from the **Edit** menu. In the **Rename** window, enter “add” in the **New name** text field and click **OK**. In the **Confirm changes** window, click **Yes**.	Verify that the **Outline** view now contains the **add** method node instead of **sumtogether**. Verify that the refactored method has been renamed to `add` in the following places: ▪ Source file: line **LINE S2**. ▪ Target file: line **LINE T3**.
24	Activate the source tab. On line **LINE S1**, place the text cursor within the `Calculator` identifier. Select **Rename identifier** from the **Edit** menu. In the **Rename** window, enter “Totalizer” in the **New name** text field and click **OK**. In the **Confirm changes** window, click **Yes**.	Verify that the **Outline** view contains the **Totalizer** class node. Verify that the `Calculator` class has been renamed to `Totalizer` in the following places: ▪ Source file: line **LINE S1** ▪ Target file: lines **LINE T1** and **LINE T2**

25	Activate the target tab. On line **LINE T1**, place the text cursor within the `source_rename_factor_test` identifier. Select **Rename identifier** from the **Edit** menu. In the **Rename** window, enter “base_rename_factor_test” in the **New name** text field and click **OK**. In the **Confirm changes** window, click **Yes**.	Verify that the source tab header has been renamed to **base_rename_factor_test.py**. Verify that the refactored module has been renamed to `base_rename_factor_test` in the target file on line **LINE T1**. Using any file browser available on your operating system, navigate to your test folder and verify that it no longer contains file named “source_rename_factor_test.py” and contains a file named “base_rename_factor_test.py” instead.

Block Comment Toggle

The test suite will use one test file, which will be created by the tester. Performing the tests will verify that enabling and disabling block comment toggle works correctly in all anticipated situations.

Testing Steps

Table 12 lists the testing steps in the order that they must be performed.

Table 12. Block comment toggle testing steps.

Step #	Actions	Expected result
1	Open or create a file with at least 5 lines of text. Perform a single left-click within the Editor area to position the text cursor on a line. Add any characters to the line so it is not empty. Remember the line contents, from now on referred to as `<data>`.	The file contents open in the Editor.
2	Choose the **Comment in** command from the **Edit** menu.	Two hashes ("#" characters) are added to the beginning of the line. Line starts with: `##<data>`
3	Keeping the text cursor on the same line, choose the **Comment in** command again.	Two more hashes are added to the beginning of the line. Line starts with: `####<data>`
4	Keeping the text cursor on the same line, choose the **Comment out** command from the **Edit** menu.	Two hashes are removed from the beginning of the line. Line starts with: `##<data>`

5	Keeping the text cursor on the same line, choose the **Comment out** command again.	Two more hashes are removed from the beginning of the line. Line contents `<data>` must now be exactly equal to that observed at step 1.
6	By holding the left mouse button down within the Editor area, select multiple lines and press **Ctrl+3**.	Two hashes are added to the beginning of all of the selected lines.
7	Keep the same line selection and press **Ctrl+4**.	Two hashes are removed from the beginning of all of the selected lines.
8	Modify a line so it starts with 3 consecutive hashes: `###` Press **Ctrl+4**.	Two hashes are removed from the beginning of the line. The line now starts with one hash.
9	Modify a line containing `<data>` so it starts with one hash, followed by a symbol other than a hash. For example: `#a<data>.` Press **Ctrl+4**.	The single hash is removed from the beginning of the line. Using the given example, the line is now: `a<data>`
10	Modify a line so it starts with a character other than a hash, followed by a hash. For example: `a#<data>` Press **Ctrl+4**.	The line must remain unchanged. Using the given example, the line is still: `a#<data>`

Appendix B: License