UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Informatics Curriculum

Rene Lehtma

Visual Aids for Programming in the Thonny IDE

Bachelor's Thesis (9 ECTP)

Supervisor: Aivar Annamaa

Tartu 2016

Visual Aids for Programming in the Thonny IDE

Abstract:

This thesis provides an overview of the development of four software plugins for Thonny, a Python IDE for beginners. All four of these plugins provide visual help with programming in Thonny.

One of the plugins is for highlighting open strings. Another is for showing the closest surrounding parentheses of the text cursor's position. These are expected to help with spotting certain syntax errors.

A third plugin is for highlighting all the usages of a name. The final plugin is to help with visually identifying if a variable is global or local.

Keywords:

syntax highlighting, IDE, programming

CERCS: P175

Visuaalsed abivahendid programmeerimiseks Thonny arenduskeskkonnas

Lühikokkuvõte:

Käesolev lõputöö annab ülevaate nelja pistikprogrammi arendusest Pythoni arenduskeskkonna Thonny jaoks. Kõik neli programmi pakuvad visuaalset abi Thonny-s programmeerimisel.

Üks pistikprogramm on lõpetamata sõnede esiletoomiseks. Teine näitab tekstikursori asukohale lähimaid ümbritsevaid sulge. Need kaks peaksid aitama teatud süntaksivigade märkamisega.

Kolmas programm on mingi kindla nime kõikide esinemiste esiletoomiseks. Viimane programm aitab visuaalselt eristada globaalseid ja lokaalseid muutujaid.

Võtmesõnad:

süntaksi esiletoomine, IDE, programming

CERCS: P175

Table of contents

1. Introduction

Learning to program can be a daunting task. The difficulty of this process can be largely dependent on the capabilities of the tools used for teaching it. One such tool is Thonny[1], an integrated development environment (IDE) for Python that has been designed with beginners in mind.

Thonny provides many helpful features for beginners, especially where comprehending program flow is concerned. These features include stepping through program code, statement by statement; visualizing recursive function calls; and viewing variable values in memory at any step of program execution. There are, however, some usability features that are common in other modern development environments that Thonny is currently missing.

This thesis aims to give an overview of a software development project where four plugins for the Thonny IDE were developed, each corresponding to a feature that would hopefully assist visually in spotting certain kinds of syntax errors and improve program comprehension. The added features are: visual distinction of open and closed string literals, highlighting of parentheses surrounding a position in the source code, showing the usages of a name within the code editor, different styling of global and local variables.

First, some background information on Thonny is given as well as motivation for expanding its current functionality. After that, the work methodology and the technologies used are described. Then, the implementations themselves are discussed in detail. Finally, a section describing the testing of the solutions is provided.

These additions are expected to improve Thonny's user experience. Some of the implemented features (e.g. name usage highlighting, global/local variable highlighting) can also hopefully make it easier for novices to understand concepts such as global/local variables and scopes in general.

2. Background and Motivation

2.1 Thonny

Thonny[1] is an IDE for the Python programming language[2]. It was created in the University of Tartu by Aivar Annamaa and is mostly intended to be used as an environment for learning and teaching programming. To support this end, Thonny offers features that are useful in demonstrating a number of programming concepts such as variable binding, recursion, debugging, and more. Figure 1 shows some of these.

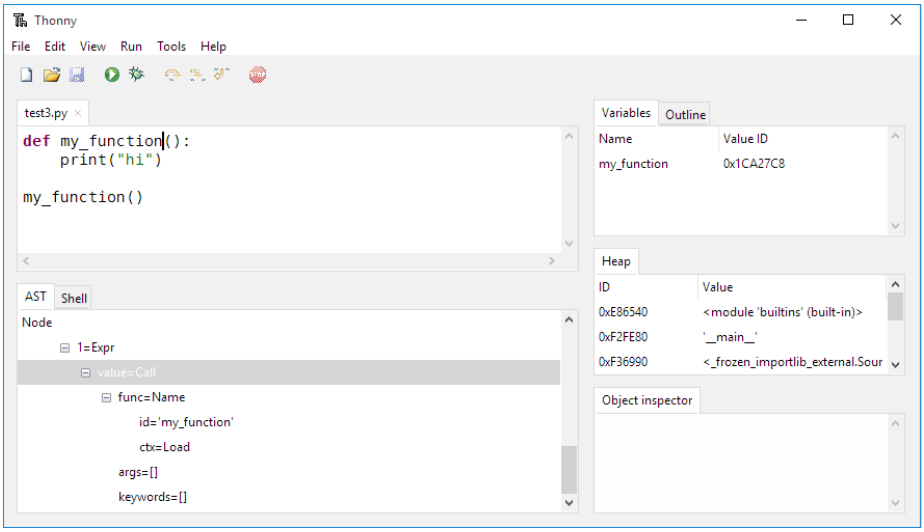


Figure 1. Thonny's Graphical User Interface. Variables and values in memory can be seen on the right.

2.2 Motivation for Additions

Many programmers can speak from experience to the benefits of syntax highlighting, but there is also evidence in research to suggest that syntax highlighting improves program comprehension[3] and that having richer syntax highlighting can be even more helpful in that regard[4]. So it makes sense to enhance Thonny's basic existing syntax highlighting functionality.

3. Development

3.1 Workflow

It was decided from the beginning that development of the new features should not interfere with Thonny's main development. Therefore, the original git repository was forked[5] and new features would first be developed in that fork. Instructions for running Thonny's main program with the implemented plugins and tests and also the list of relevant (to this thesis) source code files can be found in the repository's wiki[6]. Only once a feature was deemed to be working and stable, would it be merged with Thonny's main development branch.

3.2 Technologies Used

Thonny, including its existing plugins, has been developed using Python and uses the standard *tkinter*[7] package for displaying its graphical user interface. Plugins developed as part of this thesis will also stick to using these technologies. Additionally, two of the new plugins will make use of Jedi[8], a refactoring and static analysis library, for its capabilities in analyzing Python code.

3.3 Development of the Plugins

3.3.1 Thonny's Plugins

Plugins are a way of keeping Thonny's core application logic separate from its non-essential parts. This provides modularity and encapsulation, which reduces interdependence between the different plugins and the main application. For that reason it was logical to implement the new functionalities as separate plugins. This also makes it simple to enable or disable any of the functionalities without affecting others.

Thonny's plugins are loaded at runtime from the *plugins* package. Any python code file in the package directory that contains a function called *load_plugin*, is considered a plugin and loaded by Thonny's main program. This function is also called when the plugin is loaded.

3.3.2 Tkinter and Widgets

As stated before, Thonny's graphical user interface (GUI) is built using Python's *tkinter* package, commonly used for this purpose in many Python applications. A central concept in *tkinter*, as with many GUI libraries, is the widget. A widget is a class that represents an element of the user interface. *Tkinter* provides many widgets out of the box, but they are also easily extensible and Thonny uses several extended custom widgets.

Widgets can communicate with each other and with other code through events. Events are an integral part in how the four newly implemented plugins work. When a plugin is loaded, it creates a new object that binds one of its methods to a Notebook widget's (this is Thonny's text editor area) tab change event. A Notebook widget is essentially a box of text with multiple tabs. Due to this binding, whenever a tab is changed, this method is called.

This method then gets the active Text widget and binds another method, *on_change* to events of the Text widget that occur when something within the Text widget changes (cursor position, text is entered etc.). So now, whenever a change is made in the text editor, the *on_change* method is called. The exact nature of this *on_change* method differs between each of the plugins.

3.4 Highlighting of Open Strings

3.4.1 Description and Rationale

Since an unclosed string in the source code of a Python program is a syntax error, it can be helpful to spot these at a glance before trying to run said program. A way to provide such notification to the developer, is to just color the offending string differently. So instead of the regular green (or whatever way an ordinary closed string is colored), an unfinished string appears differently. Figures 2 and 3 illustrate this functionality.

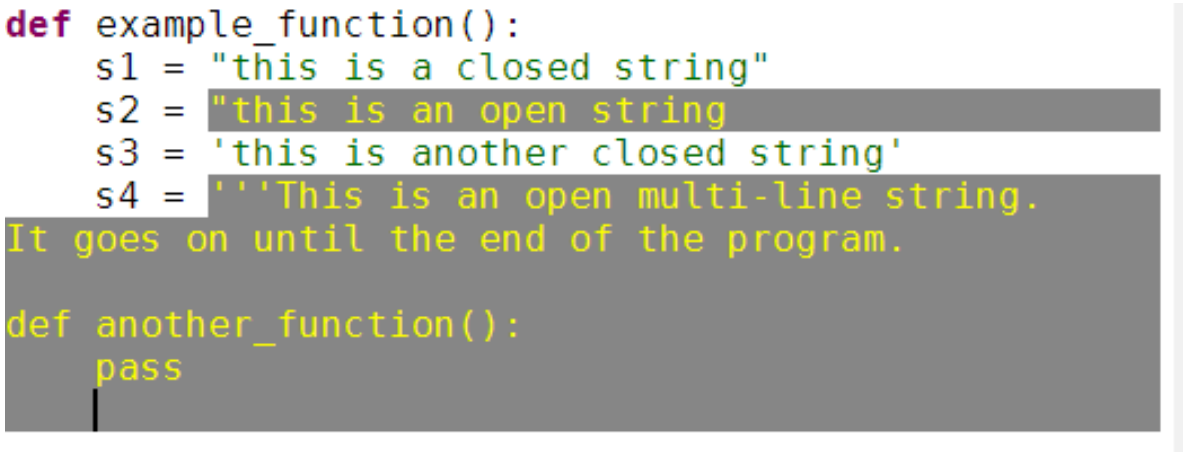


Figure 2. Four strings. Two closed strings in green and two open strings in yellow.

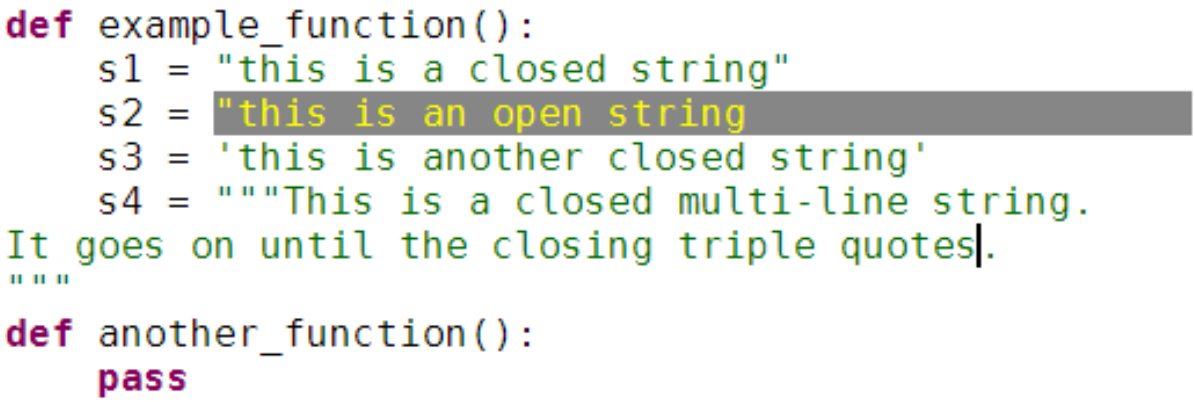


Figure 3. Same as in Figure 2, except that the last string is now closed.

3.4.2 Python's Strings

Python has several different kinds of string literals[9]. A string can be delimited by single (') or double (“) quotation marks. There are both single and multi-line versions of strings. Strings can start with certain prefixes such as 'r' and 'u', among others. Strings can also contain escaped characters. The implementation takes this into account. An escaped quotation mark, for instance, is not considered the end of a string.

3.4.3 Overview of Implementation

Most of Thonny's existing syntax coloring was (and still is) based on source code found in idlelib, a support library used by IDLE[10]. This implementation uses regular expressions as rules for finding syntactic constructs in source code. Each syntactic part has associated style information in the form of a Tkinter Text widget's tag. Each match found using the regular expressions is styled according to this style information by setting a tag at the match position within the Text widget.

This syntax coloring functionality proved simple to extend. Much of the code remained the same, although some of the regular expressions had to be modified and some new ones added along with other minor changes. A new tag style was added for open strings as well.

Keeping future maintenance in mind, the coloring functionality was also refactored into a separate plugin and removed from Thonny's main code.

3.5 Highlighting of Surrounding Brackets

3.5.1 Description and Rationale

Another common feature that many modern IDEs and text editors have, is highlighting a closing bracket matching a selected opening bracket and vice versa. A similar feature was developed in this thesis with a small modification – the closest surrounding brackets to the text cursor's position are highlighted. This can perhaps be most noticeably helpful when nesting several function calls or list comprehensions, or other nested constructs. Figures 4 and 5 show how the highlighted brackets depend on the location of the text cursor.

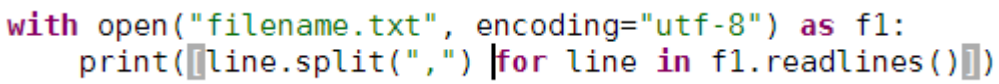


Figure 4. Text cursor before *for* keyword. Closest surrounding brackets are square and highlighted.

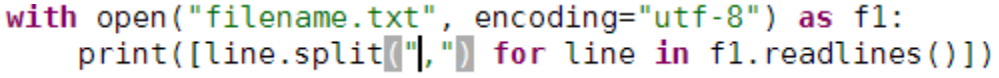


Figure 5. Cursor is now at a different position. Now the closest surrounding brackets are round.

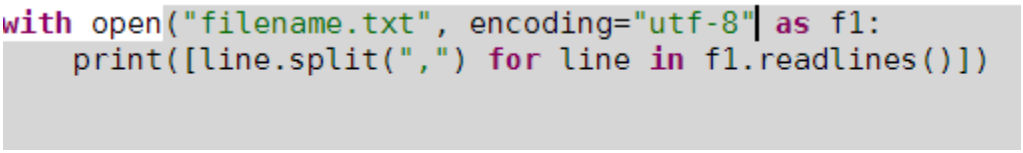


Figure 6. Imbalance. There is no corresponding closing bracket for the opener on the first line.

3.5.2 Python's Brackets

Python has three different kinds of brackets:

1) round brackets - '()' - these are used, for example, in function calls and generators;

2) square brackets - '[]' - used, for example, with list comprehensions or when accessing the elements of a collection;

3) and curly brackets - '{}' - mostly used with dictionaries and sets.

A combination of these could possibly be (and often is) nested within each other.

3.5.3 Overview of Implementation

An existing solution was examined and tried. IDLE[10]'s Hyperparser module was already included in Thonny as a dependency (though later removed) and had the necessary methods for determining pairs of surrounding brackets. This worked reasonably well, but the implementation was somewhat bulky and difficult to understand. Ultimately, in order to remove this dependency and provide a leaner and more easily maintainable implementation, the functionality was reimplemented as a separate plugin.

A description of the implemented algorithm follows. First, input is taken from a Text widget. The text string is tokenized and all tokens not corresponding to opening and closing brackets are filtered out while preserving the order of the remaining tokens.

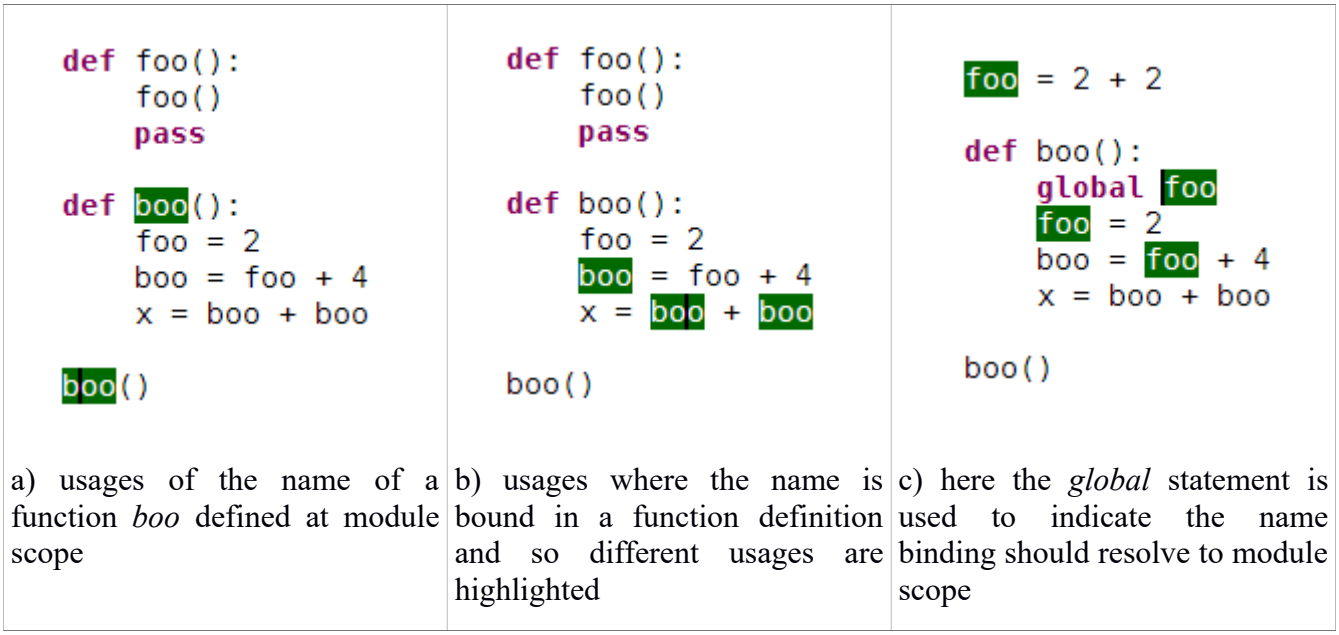
After that, each of the remaining tokens is processed in order. When an opening bracket token is encountered, it is added to the stack. Conversely, when a closing bracket token is encountered and the corresponding opening bracket token is on top of the stack, the opening bracket token is popped from the stack. If the index of the text cursor's insert position is between the opening token's position and the closing token's position, then the opening and closing tokens are surrounding the insert position, and these positions will be colored by the plugin (unless overwritten later by an opener at a deeper nesting level beacause only the nearest surrounding brackets should be highlighted).

If the stack is not empty by the end of processing all of the tokens, there is an imbalance, where an opening bracket was found with no corresponding closing bracket. In this case, everything from the unmatched opening bracket's position until the end is marked, as seen in Figure 6.

3.6 Highlighting the Usages of a Name

3.6.1 Description and Rationale

Understanding variable (or name) scope can be an issue for some novice programmers[11]. To help with this, it can be useful to visually indicate where a particular name is visible. The benefits are also apparent when trying to manually refactor code (changing the name of a variable).



a) usages of the name of a function *boo* defined at module scope

b) usages where the name is bound in a function definition and so different usages are highlighted

c) here the *global* statement is used to indicate the name binding should resolve to module scope

Figure 7. Different names highlighted depending on where the text cursor (the black vertical line) is.

3.6.2 Python's Names and Blocks

The following is a short summary of the relevant parts from the „Execution model" section of the Python language reference [12].

A Python program is made out of blocks. A block is either a module (the top-level block), a function definition, or a class definition.

To reuse a Python object, it has to be bound to a name. Names can be bound in a number of different ways: in assignment statements, class and function definitions, or a for loop header, to name a few.

A global variable is a name that is bound at the module level. A local variable is a name that is bound in a function or class definition unless the name is specified as global using the keyword *global.*

Normally, a name binding is resolved in the closest surrounding block, but the keyword *global* tells the interpreter that the name refers to a global binding. Names bound in a surrounding scope can be used within an inner scope, but only if the name is not bound in that inner scope. If a name is bound in a block, it cannot be used before it is bound.

3.6.3 Overview of Implementation

Jedi[6] is an open-source refactoring and static analysis library for Python. It has some built in functionality for finding the usages of a name in Python' source code. Unfortunately, several problems were identified when trying to make use of this existing functionality. For one, when searching a name's usage in a single-argument function call, nothing was found when using this existing functionality, nor were the proper associations with global variables made in all cases.

Since these issues did not have anything to do with Jedi's parser itself, it was decided to base the implementation on processing the abstract syntax tree (AST) that Jedi's parser produces from Python's source code. Some consideration was also put into using Python's *ast* module, which also provides a parser. Jedi's parser had one determining advantage over the *ast* parser - the ability to handle broken syntax.

Here is a basic description of the algorithm used for finding the usages of a name:

First, a definition for the name is searched for. For that, the closest surrounding scope (block) is searched for a definition. If not found there, the next surrounding scope is searched and so on. If after searching through module scope, still no definition is found, then all usages of the name will be highlighted, regardless of scope.

If a definition is found, then only the usages of the name in scopes where this binding in particular is used, will be highlighted. For example, if a subscope of the names definition's scope binds this name to another definition, the usages in that scope will not be highlighted.

It is not ideal that this implementation relies on Jedi's AST structure, which is subject to change as it is not part of its public API. This means that if Jedi were to be updated in a future version of Thonny and changes had been made to Jedi's tree structure, it would be necessary to also update these plugins to account for those changes. The AST is likely to change if major changes are made to Python's grammar.

3.7 Highlighting of Global and Local Variables

This section is fairly short, because much that was discussed in 3.6 applies to this functionality as well.

3.7.1 Description and Rationale

It can be helpful to tell at a glance, which variables in the code are global and which are not. Displaying these in different styles or colors makes the distinction apparent.

```
foo = 2 + 2

def boo():
    global foo
    foo = 2
    boo = foo + 4
    x = boo + boo

boo()
```

Figure 8. Here, local variables are underlined and globals are not.

3.7.2 Overview of Implementation

This plugin's implementation consists of two parts:

1) finding the locations of all the global and local variables in the program and storing them in their respective lists;

2) coloring these locations according to the style information configured in the plugin.

The first part relies on Jedi's AST and traversing it recursively, starting from the module scope. Names are added to the global or local lists according to the name binding rules described in section 4.4.2.

The second part is simply setting the correct style tags according to positions found in the first part.

4. Testing

4.1 General

Unit tests were written for each of the four plugins. All of the tests follow a similar pattern. A Text widget containing Python code is created. The Text widget is then given to a plugin's function implementing its specific functionality. With the string coloring functionality, the Text widget is then checked to see if the correct styling tags are set in the right places. With the other functionalities, output from the plugin's main working function is checked. This output is in most cases a set of tuples with Text indices in the form of *l.c*, where *l* is the line number and *c* the column number.

4.2 String Highlighting Tests

There are test cases for checking highlighting in code containing both open and closed strings and both single- and multiline versions of those as well as multiline strings containing singleline strings and also strings with escaped characters.

4.3 Bracket Highlighting Tests

There are test cases where the different kinds of brackets are nested and test cases with different bracket balancing: a case where there are more opening brackets and a case where there are more closing brackets.

4.4 Name Highlighting Tests

Here, it makes the most sense to test code where the same name is bound in different scopes. This way it can be easily checked if the correct binding of a name is highlighted. For example, when a name is bound in both module level and in an inner function definition and the Text cursor is placed on a name's usage in the module scope, none of the name usages in the function definition should be highlighted. This was also illustrated by Figure 7 in section 3.3.

For testing the global/local highlighting functionality, the test cases are very similar.

5. Conclusion

As part of this thesis, four syntax highlighting features were added to the Thonny IDE: highlighting of open strings, highlighting of brackets surrounding the text cursor, highlighting a name's usages, and highlighting of global/local variables. Each of these additions was developed as a separate plugin to keep the functionalities independent from each other as well as to provide ease of maintenance.

All of the plugins were developed using the Python programming language and tools from its standard library. Two of the plugins – those responsible for highlighting a name's usages and highlighting global/local variables – required deeper parsing of Python code. For this, the parser from Jedi, a third party static analysis library, was used.

Each plugin is accompanied by unit tests covering its common use cases.

These new functionalities are expected to help with spotting certain syntax errors, to help with program comprehension and to provide an overall improvement for Thonny's user experience.

References

[1] http://thonny.cs.ut.ee/ University of Tartu. Institute of Computer Science. Thonny. (2016-05-11)

[2] https://www.python.org/ The Python programming language. (2016-05-11)

[3] Sarkar, Advait. "The impact of syntax colouring on program comprehension." *Proceedings of the 26th annual workshop of the psychology of programming interest group (ppig 2015)*. 2015.

[4] Asenov, Dimitar, Otmar Hilliges, and Peter Müller. "The Effect of Richer Visualizations on Code Comprehension." *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 2016.

[5] https://bitbucket.org/rene_lehtma/thonny_fork Development fork for thesis plugins. (2016-05-11)

[6] https://bitbucket.org/rene_lehtma/thonny_fork/wiki/Home Repository's wiki page. (2016-05-11)

[7] https://docs.python.org/3.5/library/tkinter.html Tkinter, Python interface to Tcl/Tk. (2016-05-11)

[8] http://jedi.jedidjah.ch/en/latest/ Jedi - an awesome autocompletion/static analysis library for Python. (2016-05-11)

[9] https://docs.python.org/3.5/reference/lexical_analysis.html#string-and-bytes-literals The Python Language Reference. String and Bytes Literals. (2016-05-11)

[10] https://docs.python.org/3.5/library/idle.html IDLE. Python's Integrated Development and Learning Environment. (2016-05-11)

[11] 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.

[12] https://docs.python.org/3/reference/executionmodel.html The Python Language Reference. Python Execution Model. (2016-05-11)

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Rene Lehtma,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

„Visual Aids for Programming in the Thonny IDE”,

supervised by Aivar Annamaa,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **09.05.2016**