Gottfried Wilhelm
Leibniz Universität Hannover
Faculty of Electrical Engineering and Computer Science
Institute of Practical Computer Science
Data Base and Information Systems Section

Investigating and Improving Variable Names in Data Science Projects With Data Mining

Master Thesis

in Computer Science

by

Huu Kim Nguyen

First Examiner: Prof. Dr. Ziawasch Abedjan
Second Examiner: Prof. Dr. Marius Lindauer
Supervisor: M.Sc. Binger Chen

Hanover, May 9, 2022

Declaration of independence

I hereby certify that I have written the present master thesis independently and without outside help and that I have not used any sources and aids other than those specified in the work. The work has not yet been submitted to any other examination office in the same or similar form.

Hanover, May 9, 2022

Huu Kim Nguyen

Abstract

Data science is a growing interdisciplinary field, but their naming conventions are slightly different than the conventions used in regular Python code.

Descriptive variable names lead to self-documenting code, which improves code readability and code quality. Many variable names in data science projects are abbreviated or have a single letter, which are difficult to understand for developers.

To expand abbreviations that have more than one character, a lookup tree data structure was created from data mining 20925 GitHub repositories that used Python to find lines that contain abbreviated variable names and their full word. It can also fix the naming style of the name to conform naming conventions and shorten long names. Replacing single-letter variables data mines for any statements within a function in which the variable is used to find its usage context. Both were based of previous algorithms.

The claim that data science code has a lower quality was tested against a non data science dataset. While more similar than expected in terms of code quality, slightly winning and losing out in different aspects, data science projects do use slightly more single-letter variables and are more likely to use non-standard indentation spaces.

The abbreviation expansion algorithm has an accuracy of 65,75%, which is a minor improvement. The single-letter replacer has an accuracy of 27,17% because of differences in the programming language from which the idea originated. The integrated development environment plugin, once the dataset creation and installation is finished, can resolve a name in less than two seconds.

Table of contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The field of data science is growing in popularity and it is growing in importance in businesses [1]. Data scientist jobs are still in demand in 2021 and have a job growth of 28% through 2026 as estimated by the U.S. Bureau of Labor Statistics [2], even with the advent of artificial intelligence which can be used by data scientists. It was called “the sexiest job of the 21st century” in October 2012 by the Harvard Business Review [3].

The Python programming language is the most used programming language in data science projects in GitHub [4] and is part of the Jupyter Notebook application which incorporates Python and is based of the IPython interactive interpreter [5]. This programming language is also the most popular programming language in general as of October 2021 [6], and is used by Facebook, Netflix and Google which are considered to be part of the Big Five tech companies [7]. Therefore it is expected that both Python and data science will remain popular in the near future.

However, data science projects have something of a bad reputation among programmers [8]. Even if said claims are anecdotal, these claims exists for a reason, otherwise we would hear anecdotal praises of them if the opposite happened. One area which is also claimed to be poor is in variable naming conventions [9]. Caprile and Tonella said that “Identifier names are one of the most important sources of information about program entities” [10]. Using a consistent variable naming convention will lead to a better overall code quality which leads to code that is more understandable by the programmer and is less ambiguous [11][12]. To ask the other way around, there is no reason not to have a better code quality and to have a better code quality requires better variable names.

A higher code quality also has a higher market value [13]. Less time is spent figuring out what the code means and more time is productively spent into programming, which also means that deadlines can be met sooner. There is the concept of “technical debt”, which are additional “costs” that are caused by poor software quality, usually referring to the time that is lost. As an analogy, a “shortcut” solution is taken to save time in the short-term to meet a tight release deadline, but doing so will hurt in the long run when the software needs to be maintained, because the “shortcut” causes worse code quality [14].

There are multiple ways to improve code quality [15], such strictly following guidelines to ensure that a single function does not have an excessive cyclomatic complexity [16] by limiting the number of branching and loops. The number of nested blocks can be reduced too. Another way to improve code quality is to improve the cohesion [17] by adhering to the single-responsibility principle and lower the coupling between code modules, which is also achieved by reducing its dependencies between the modules [17]. There is also the aspect of documentation which can be automated [18]. Even with all of that, there is one aspect which is always present and cannot be ignored, which is the naming of variables.

One way to improve the code quality is to improve the variable names, which was not done in Python so far and this process can be automated. As already mentioned, these names are a very significant piece of context regarding the code [10]. Using more descriptive variable names instead of abbreviations and specialized terms that only certain programmers can understand leads to code that is easier to read. From this thesis, data science projects use slightly more abbreviations and more single-letter variables than non data science projects.

The goal of this thesis is to create a plugin for an integrated development environment (IDE) which replaces abbreviations and single-letter variables with full English words which are more descriptive. This not just improve the code quality but also allows the programmer to check what the abbreviation actually means.

1.1 Potential users

The primary target group of users are data scientists, especially aspiring ones, that work with such code as described in the previous sections. Data scientists often use different and specialized terminology that is not used in any other fields of programming and are therefore unknown to other programmers [9]. Such a tool that would highlight what tat terminology actually means would help newer programmers understand their terminology better and replace abbreviations with more meaningful full English words.

Other programmers that are not data scientists can benefit from such a tool too. The tool works in a general fashion which is not too specifically tailored for data scientists. As such, it can be applied to other projects as well. The benefits would be the same as above. Not only can better variable names lead to a better code quality, but better code quality can be also more valuable, as less time is spent guessing what the meaningless variable name means and more time can be spent doing productive work [13].

1.2 Challenges

There are multiple challenges that new developers face when looking at data science code.

- Examining code from these projects is difficult without prior knowledge.
- Understanding the abbreviations used in data science projects.
- Time required to learn abbreviations that could have been used productively.
- Risk of schedule overruns due to the time lost [19].
- Possible lack of proper documentation or in certain (private) functions.
- Code may not be not self-documenting [20] due to meaningless variable names.

This thesis tries to tackle the issue of abbreviated variable names and single-letter names by expanding them into full words, which are more meaningful. Doing this will solve the challenge of having to learn these abbreviations since their full words are used instead of the abbreviations. It also indirectly solves some other challenges, at least partially. As time is not spent learning abbreviations, it can be more productively, such as programming the code which also lessens the risk of missing a deadline. Well written variable names also partially contribute towards a self-documenting code [20], assuming the function is clean.

That being said, self-documenting code does not fully replace actual documentation which is outside the scope of this thesis. Automatic documentation already exists [18]. Even with better names, if the rest of the code is hard to read the code quality will remain questionable. Other techniques are required, such as lowering the cyclomatic complexity [19], decoupling any classes, and ensuring a high cohesion, which are outside of the scope of this thesis.

1.3 Proposed solution

The solution is to expand meaningless variable names into more verbose, more understandable, more meaningful variable names which are easier to understand by developers.

Large amounts of Python code needs to be collected and analyzed, to get a consensus of what variable name used in a given context. Two different approaches are used depending on the length of the variable name. An English dictionary is required in both approaches to check for English words.

If the variable name has at least two alphabetic characters, then it is assumed to be an abbreviation, which is then expanded. The large code dataset is then used to look for any hints how these abbreviations could be expanded into full English words by looking at the statement in which the variable is assigned from other variables or functions to create a lookup tree. This idea was based of a previous work by Lawrie et al. [21], but it only accounted for local code in a single file and comments and it did not work on Python, given the technical limitations of that time. This new approach uses a large dataset to resolve abbreviations. It also implements type inference as an additional piece of context. This new work also gives the option to choose from three choices, instead of one.

If the variable name does not have at least two alphabetic characters, then it is treated as a single-letter variable, which is not an abbreviation and is too ambiguous to be expanded by the previously mentioned abbreviation expansion. The idea here is to rather replace that one outright with a different variable name. Instead it looks at how the variable is used within the context in a single function or in other words how it is used in a function. This was based on a previous work by Tran et al. [22], but it was very specifically written for JavaScript and not for Python, which means it needs to adopted to work in Python and is written do rely less on specific Python features. Both new ideas should be language-agnostic.

These new approaches create resources that are required for a plugin that will be implemented in an integrated development environment (IDE), which can be installed by the user. These resources can be thought up as a large lookup table, meaning that while the computation of these resources can take a while, the data on the plugin can be looked up quickly, up to two seconds. The plugin then tries to resolve that variable name using the resources and the user is then presented with three choices, the best one being highlighted. After choosing one choice the variable name is then replaced with a new one. For the user, this is as easy as right-clicking the variable name, then choosing to replace that name.

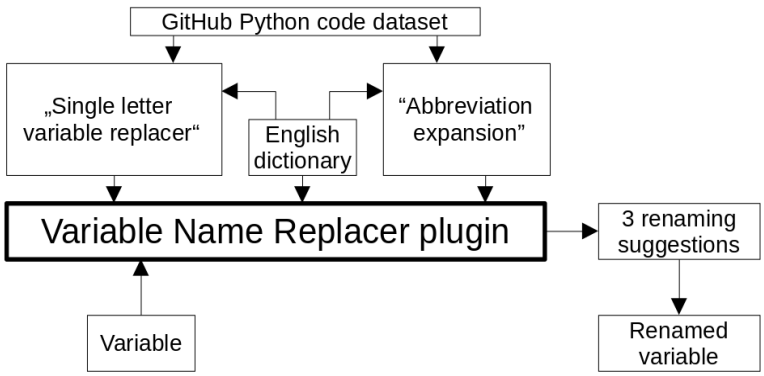


Figure 1.1: Basic structure of proposed solution

Figure 1.1 shows the basic architecture of the solution. Both algorithms, the “Single letter variable replacer” and the “Abbreviation expansion”, that require large set of GitHub repositories that use Python and an English dictionary create resources for the Variable Name Replacer plugin. That takes one variable to suggest three replacements. The user picks one to replace the name.

There is one example in Listing 1.2. It shows one line that does not use full descriptive words and one line that does. The line with the full words is easier to understand, while the one with abbreviations takes a while to comprehend, especially if they are not familiar with the abbreviations. The formula is the volume of a cylinder [23].

```
v = rd * rd * pi * h                    # abbreviations
volume = radius * radius * pi * height # full words
```

Listing 1.2: Example regarding descriptiveness of names

1.4 Structure of the thesis

The first chapter, as already mentioned before, describes the current situation regarding this thesis, the problem which can then be solved with the proposed solution. It also mentions any potential users.

Chapter 2 describes the research questions of the thesis and how they will be prioritized during the creation of the thesis.

Chapter 3 explains some necessary but already well established basic concepts that are required for the newer concepts that are introduced in the next chapters. Variable names and naming are explained again, alongside with coding and naming conventions in the Python programming language. The verbosity of variable names will be explained and so are type hints and type inference.

Chapter 4 introduces some related work that is incorporated in the later chapters. It first briefly discusses about the current state of type inference in Python which is still imperfect. After that, algorithms regarding variable name recovery are mentioned. It also mentions existing attempts of expanding abbreviations into full English words. Other related work includes research on how variable name length affects readability and what their results mean and previous research on code quality in data science projects.

Chapter 5 is split up into three parts which describe each theoretical concept. Before any of that, the broad architecture of the concepts and how they interact are explained as well as a brief explanation on why an English dictionary is required. The first concept describes how the Python code dataset is created and used for the later steps. The second concept is the full explanation of abbreviation expansion, including the data structures required for it and any additional pre- and post-processing required for that concept. The third concept is the full explanation of single-letter variable replacement, which works differently than the previous concept.

Chapter 6 shows a technical implementation in form of an IDE plugin. It describes the full setup, including any prerequisite resources that are required, and usage of said plugin. The end-user architecture in the plugin itself is also explained.

Chapter 7 is the full evaluation. It describes the initial evaluation setup and tools. It looks at what kind of variable names are used in these datasets and how it impacts the code quality. Then the accuracy of both the abbreviation expansion are examined. Finally the time performance of the algorithms are investigated.

The final chapter of the thesis consists of the summary of the thesis. After that there is an outlook which describe some potential improvements that have not been implemented yet and the conclusion of this thesis.

Chapter 2

Research questions

The second chapter is split into Section 2.1 which details the six research questions that are answered in Chapter 7 and into Section 2.2 which sets the order on which research questions are answered first.

2.1 Research questions

[RQ1] What naming conventions in current python code style guides do data science projects comply / not comply with?

The first question questions whenever data science projects comply with current Python naming conventions [24] or not. If they violate any of the conventions which exact ones do they violate? And what conventions are followed properly?

[RQ2] How can we efficiently mine these patterns in a big code repository?

This is mainly a question on the performance of a program that can data mine thousands of data science projects. How fast can that program process that many lines of code? Many external factors can also play into this, such as the type of hard drive, the file system and the operating system.

[RQ3] What is the difference between data science projects and other projects regarding naming conventions?

Are there any differences in how data science projects name their variable names versus other projects and if there are any, how does that contribute to the differences in overall code quality?

[RQ4] Is there relevance between these naming conventions and other code features?

Do the differences in the naming convention in data science projects even go as far as affecting the rest of the code? Are certain code features more likely to be found in data science projects, such as a different number of members, functions and modules?

[RQ5] How effective are the existing methods for general projects to generate variable names, on data science projects?

If any existing methods exist that can automatically generate variable names in Python, how good are these methods and how do these compare to the new method?

[RQ6] How can these mined naming conventions improve the variable name suggestion?

After mining many variables, can these result to better name suggestions? And how does that compare to previous solutions?

2.2 Priority

Some research questions depend on other research questions to be answered first before they can be answered. Either way the research questions have to be answered sequentially. The order in which the research questions will be answered will be stated below.

[RQ1] can be answered first, as this does not depend on anything else. A dataset of data science projects would be data mined to see if these comply to naming conventions.

[RQ2] is more of an optimization problem than a research question. Optimization is something that is done after initially writing the program and testing said program that it works at all, as opposed to premature optimization which should be avoided. Note that optimization may also require a slower computer for additional testing because some unoptimized routines only show up on a really slow computer. One thing to also watch out for is memory usage as some performance improvements come at the cost of increased memory usage. It can be done simultaneously with [RQ1] and [RQ3].

[RQ3] also requires data mining a comparative set of non data science projects so that it can be compared to a dataset of data science projects. Otherwise the same steps of [RQ1] are done but with a non data science project, then the two datasets will be compared to each other in terms of naming convention compliance.

[RQ4] requires that the naming conventions of data science are already examined to begin with. This requires a deeper view into the code of said data science projects than the previous research questions and should therefore be done after that.

[RQ5] and [RQ6] are the last research questions to be answered. [RQ5] depends on the very existence of methods that can automatically generate variable names for variables. If these do not exist at all then [RQ5] cannot be answered in any meaningful way. Otherwise they will be applied on a dataset. If multiple such algorithms exist then they would be compared against each other.

[RQ6] is the last and by far the most difficult research question. This requires the creation of a new program that can automatically generate better, more meaningful variable names and replace older variable names that were not human readable. This is still an open field of research.

Chapter 3

Basics

This chapter explains basic concepts that will play a role in the later chapters 4 to 6. It will explain basic data structures, some naming conventions in Python and how much detail a variable name can have. Some basic concepts are specific to the Python programming language and may not apply to other programming languages.

This chapter does not explain specific terminology and concepts that are only required for the evaluation, these will be explained in Chapter 7 instead. All basic concepts in this chapter were invented by different authors and are attributed and credited as such.

3.1 Variable names

Variable names are the names of so called variables which can hold a value in the program code so that the value can be referenced later in the code [25].

Most, but not all [26] high-level programming languages allow for naming variables and do not impose any limitations regarding variable name length. In most cases the only limitations are that the variable name must be written as a single string without any whitespace in-between the variable name and they typically only allow alphanumeric characters and the underscore character. A variable name must not start with a number.

Variable names are declared in the variable declaration statement, in which a value is assigned to a variable. Said variable also receives a name during this process. In statically typed languages an exact type must also be specified [27]. The most common types being integers, floating point numbers, characters, boolean expressions, and strings which are usually referred to as int, float, char, boolean, and String, respectively, but this varies slightly with each different language. One example is seen on Listing 3.1.

```
int variable_name = value; // C language
variable_name = value      # Python language
```

Listing 3.1: Variable declarations in different languages

In Python or in JavaScript, which is a dynamically typed language, no type must be specified [28]. Instead, the interpreter of that language tries to guess what type the variable uses which does decrease the time performance of the program. Some languages support type hints, in which a type can be assigned to a variable like in statically typed languages. These are to give the programmer, not the program, a hint what type is used.

Similar but not identical are the names of parameters, which are declared at the function statement. Inside the function, these functionally behave like variables. For the purpose of this thesis, variable names and parameter names are treated identically.

In other literature these two each are a subset of what are called identifiers [21] which also includes function and class names as they are used to identify these functions and classes. If the term identifier is used, either in this thesis or in a related work, in this context it means a variable or a parameter name and are treated as synonymous.

3.2 Priority queue

The priority queue [29] is a data structure which can be imagined as a list, but a list that is always sorted when a new entry is inserted to that queue. The elements inside such a priority queue must be comparable. Examples are the lowest or highest number or the alphabetic order of strings.

There are two types of priority queues, a min priority queue and a max priority queue. A min priority queue is sorted by the elements with the lowest value first, while the max priority queue is sorted by the elements with the highest value first.

At the bare minimum, four operations are required.

1. The insertion of an element to the priority queue.
2. The retrieval of an element from said priority queue, without altering the queue.
3. The same as the second operation but said element is also removed from the queue.
4. A basic check if the queue is empty or not.

The exact implementation can vary as long as these operations are fulfilled. While it can be done in a linked list, a more time efficient way to achieve this is by using a heap [30], having a worst case time complexity of only O(log n) in typical operations such as insertion and deletion.

3.3 Tree data structure

A tree is a data structure that is ordered using a tree analogy [31]. These data structures always start with a “root”, with “nodes” in-between and always end with “leaves”.

The relation between nodes can be defined analogous to parents and children. For an example as seen in Figure 3.2, “leaf 1”, “leaf 2”, and “leaf 3” which are “siblings” are the children of “Node A”, who is the parent of these “siblings”. All nodes which have no “children” are “leaves”. The depth of a node is the number of edges from the root to the node. “Node A” has a depth of one.

Unlike lists, which are linear data structures and require a linear amount of time for its operations, trees are not linear.

In a different example that is used in decision trees [32], there are two categories which are used to separate entries. These two categories are the human gender and any persons that have an age of at least 25 years, respectively. Using these two categories, the nodes in depth one are “male” and “female”, the nodes in depth two are “age >=25” and “age <25”. The leaf nodes are each human entry that meet the criteria of the parent nodes. The number of leaf nodes under each parent node are always smaller than putting all leaf nodes into one large list. Using a decision tree it is possible to find these leaf nodes faster than searching for an entry in a list. This only describes a generalized tree data structure.

There are many different specialized tree data structures which have their own set of operations and have additional rules on how these trees are structured, but the basic idea of a tree structure is always given.

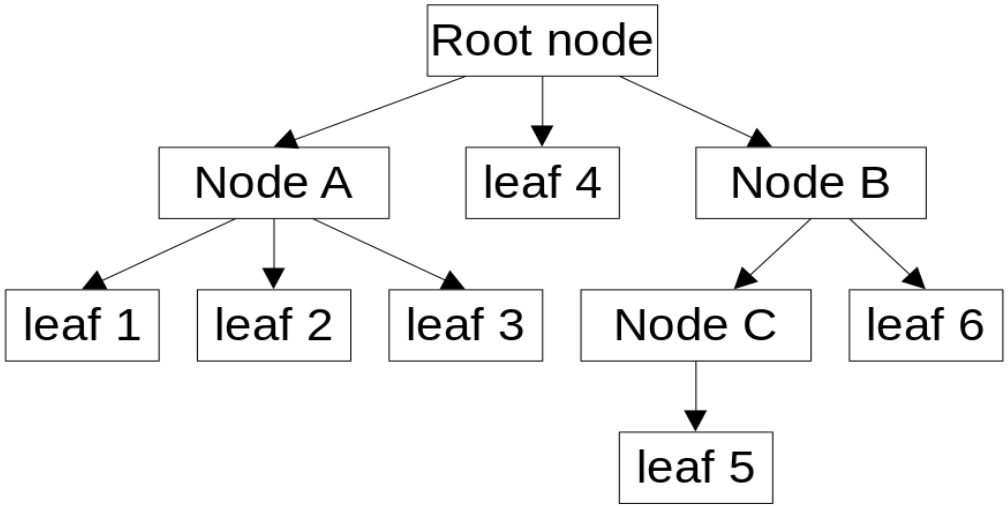


Figure 3.2: Example tree data structure

3.4 Serialization

Serialization is the process of turning an internal data structure that is executed within code to a bitstream that can be stored outside of the program [33], typically on a hard drive. The data structure is only defined inside the confines of the program. Serialization is required if the data structure "leaves" the program, for a variety of reasons. Not just to store said data structure on the hard drive or in a database but also to transfer the data structure to a different computer in a different place over the network for another example. This idea is depicted in Figure 3.3.

One important counterpart is deserialization, or the reverse process of turning said bitstream back into the internal data structure. This is required when the data structure needs to be loaded back from the hard drive to the program or when the recipient takes the serialized bitstream and needs to convert it back to the internal data structure.

The human readability of the bitstream is not required and the serialized output can be compressed to reduce the size of the bitstream. If it is compressed, then the bitstream needs to be decompressed in the process of deserialization.

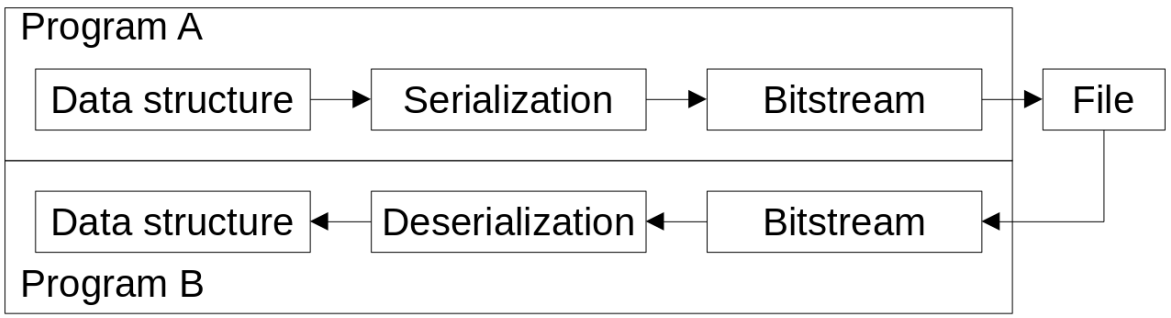


Figure 3.3: Simplified serialization and deserialization

3.5 Naming conventions

Naming conventions are a subset of a coding convention which differ slightly with every different programming language. At the most basic level it is required because variable names cannot have any whitespace characters so that the variable can be detected as a single word in the interpreter. For the Python programming language, there is a subsection dedicated to naming conventions in PEP 8 [24] which is summed up in Table 3.4.

Types of names	Style	Example
Constants	All-uppercase, separated by underscores	CONSTANT_NAME
Variables	All-lowercase, separated by underscores	variable_name
Global variables	All-lowercase, separated by underscores	global_variable_name
Functions	All-lowercase, separated by underscores	function_name
Methods	All-lowercase, separated by underscores	method_name
Modules	All-lowercase, separated by underscores	module_name
“Public” attributes	All-lowercase, separated by underscores	public_attribute_name
“Private” attributes	All-lowercase, snake case with _ at start	_private_attribute_name
Type hints	PascalCase, starting with capital letter	TypeHintName
Classes	PascalCase, starting with capital letter	ClassName
Exceptions	PascalCase, with Error as suffix if it is one	ExceptionNameError
Not for subclass	All-lowercase, snake case with __ at start	__super_variable_name
“Magic” methods	Same but with leading and trailing __	__init__, __new__

Table 3.4: Naming conventions in Python per PEP 8 [24]

Usage of non-ASCII characters is strongly discouraged and not allowed at all when using standard library variables. All names should also not be shorter than 3 characters and not longer than 30 characters.

A single underscore as postfix after the name can be used to name variables when the actual name is the same as a reserved keyword. For example, if the programmer wants to name a variable “yield” which is normally disallowed for being a keyword, a trailing underscore can be added so that the variable is allowed again, turning it into “yield_”.

Instance methods should always use “self” as the first argument. Technically an instance method always requires one argument, the first argument being used to access instance variables, but any name is possible instead of “self” for the first argument.

“self” name is used as part of the naming convention. This is initially used in the “__init__” method to create instance variables. Said instance variables can be accessed in other instance methods by prepending “self.” to the original variable name.

In class methods which must be declared with the “@classmethod” function decorator, the first argument should be named “cls” instead which is used to access static class variables. While there can be many instances per class, only one class per class can exist. The “@staticmethod” function decorator also exists. If that is used, then neither instance nor class variables can be accessed with that function and therefore one less argument is required. “self” and “cls” are not keywords in the Python language.

As an analogy, instance methods in Python are non-static methods in Java.
Class methods in Python are static methods with static variables in Java.
Static methods in Python are static methods without static variables in Java.

“Public” and “private” attributes are not implemented on a technical level in Python, meaning it is possible for another class in a different module to access a “private” attribute from a different class. This is solely to inform that a “private” attribute is for internal usage only and should not be used outside of the class.

To prevent a naming conflict between a subclass and its superclass if both define the same name, the superclass can have variable names with two leading underscore characters. It causes name mangling. This variable in the superclass can still be accessed though, using a leading underscore followed by the class name, followed by the full variable name. One such example is “__Class_mangled_name”. Therefore, this approach cannot be used to hide variables and make them “private” like in some other object-oriented programming languages. This practice is commonly referred as “dunder” and is somewhat controversial among Python programmers.

So called “magic” methods use special reserved names meant for the Python interpreter. The user should not define anything else with that pattern. Examples used by the interpreter are “__new__”, “__init__” or “__del__” [34]. These names must never be renamed under any circumstances or else the program will stop functioning.

Constants are not implemented on a technical level in Python. There are no keyword for constants. Because of that, their names should be capitalized to tell them apart from any other variable names. This also means that their value could be changed during runtime. Non-global variables that are outside of a class or function are considered to be constants.

3.6 Homoglyphs

Two characters are called homoglyphs if these look very similar to the human eye and therefore are difficult to tell apart from a human eye perspective, but not from a technical standpoint as these two characters use different characters [35].

For an example, PEP 8 [24] explicitly discourages the usage of the lowercase character l, the uppercase character O, and the uppercase character I. The reason being that the lowercase letter l looks too similar to the number 1 or the uppercase letter I or that

Certain fonts can worsen the effects of homoglyphs, such as the number 1 and the lowercase character l in serif fonts or the uppercase character I and the lowercase character l in sans-serif fonts.

Most programming tools use sans-serif fonts which are also monospaced [36], meaning every letter requires the same amount of horizontal space on the screen. It means that in an IDE, any multi-letter homoglyphs such as the two letters “rn” which looks similar to the letter “m”, would be correctly displayed as two letters and one letter respectively, therefore avoiding confusion, unlike in this text. Some fonts also have a slash inside the number 0 so that the number 0 can be easier to tell apart from the uppercase letter O. For that matter, any variable, function or class names cannot start with numbers and names and these must always include one alphabetic character. As such, no variable can be named after the number 0 or the number 1. Otherwise every number could be a potential variable if that rule did not exist. It is however possible, to use a leading or trailing underscore character to circumvent that rule. So the previously stated discouragement of using these letters are still valid.

3.7 Verbosity of variable names

The verbosity of variable names can be categorized into six categories [37] in Table 3.5:

x	Single-letter variable	src_fn	Multi-word abbreviation var.
x1	Pseudo single-letter variable	source	Full word variable
src	Abbreviation variable	source_filename	Multiple full word variable

Table 3.5: Examples of variable name verbosity

3.7.1 Single-letter variable

Single-letter variable names only contain a single-letter, such as x, y or z. As such no English word can be a single-letter variable and vice versa. It cannot consist of a number because variables are not allowed to start with a number. They contain the least amount of information.

3.7.2 Pseudo single-letter variable

Pseudo single-letter variables consist of two characters, the first character is always an alphabetic character as described above in the previous subsection followed by a single number. This is often used if the single-letter variable is already taken. It is not a single-letter by definition but since it is almost as meaningless as a single-letter variable, it can be treated like a single-letter variable.

3.7.3 Abbreviation variable

Abbreviations [38] must consist of at least two alphabetic characters and must not be a full English word. This means that this cannot be found in any English dictionary.

Abbreviations are formed from the original full English word or words by removing select characters from the word while trying to retain as much information as possible so that the abbreviation is associated with its original full English word.

Therefore all characters in an abbreviation must be present in the full word from which the abbreviation is derived. Both abbreviation and full word must start with the same character. Note that multiple words can form a single abbreviation.

Acronyms while having the same goal, are formed differently. An acronym is based of multiple words. From each word, the first character is taken to from the acronym. For the purpose of this thesis, abbreviations and acronyms are in the same category.

It conveys more information than single-letter variables but less than full English words. Short abbreviations can be confusing to those that are not familiar with the terminology used in that field.

There are a few acronyms whose acronym is better known than its full name, in which case the acronym should always be used over the full name when naming variables. For another example, the acronyms “URL”, “API” or “PDF” are better known than their full names “Unique Resource Locator”, “Application Programming Interface” or “Portable Document Format”. It is certainly easier to memorize these acronyms than the full names.

There are also brand names and trademarks, which would count as full English words, but are not present in any English dictionary.

One problematic aspect of abbreviations, especially shorter ones, are they can be ambiguous. For an example, the abbreviation “fn” can either mean “function”, “false negative” or “file_name”. Additional context, such as the type or the function it is located in would be required to further determine the exact word from that abbreviation.

3.7.4 Multi-word abbreviation variable

Multi-word abbreviation variables are multiple abbreviations or a combination of abbreviations and full English words, separated by the underscore character. Each individual abbreviation can be resolved individually. Otherwise the same rules as the singular abbreviation variable apply.

3.7.5 Full word variable

Full variable names that contain full English words are words that are not acronyms and can be found in an English dictionary. It can convey more information than an acronym, provided that the acronym is not obvious.

While rare, it is possible that the same English word can convey two different meanings. For an example, the animal known as the bat shares the same name as the sporting equipment known as the bat [39]. Additional context can be added to differentiate them, in the former case, using the full animal name specific to the subspecies or in the latter case, specifying in which sport that equipment is used.

3.7.6 Multiple full word variable

It is possible to add more information to the variable name to increase the verbosity. The main advantage is that it is unambiguous but the main risk here is that the name can become too long. Even if the name itself does not exceed 30 characters in length, any line that contains that variable could exceed the 80 character limit of that line. In this case, additional reformatting of the code would be required.

Complete descriptions of the variable name should not be written in the variable itself but rather in the documentation, as a full description be too long for a variable name. On that note, the internal documentation in the code usually only fully describe the parameters and the return value and not any internally used variables. Comments can be used to describe these but neither documentation nor comments are guaranteed in the Python code since the code can work without any documentation or comments whatsoever.

3.8 Generic variable names

The three single-letter variables i, j, and k are very frequently used as loop indices. This may have originated in Fortran [26] when it comes to programming languages, which in its earliest iterations had a very short maximum variable length of six characters and emphasized brevity over verbosity. If the variable name starts with one of the letters i, j, k, l, m, n then it is an integer while otherwise it is a real variable in Fortran [40]. This may also explain why these variables may have been kept short at that time.

However, the usage of the index i and k, which are used in the typical summation notation goes back even further. Possibly the first documented usage was attributed by Euler, who also was the first to use the sigma character to denote summations [41]. The names x, y, and z which are attributed to Descartes [42] who also invented the Cartesian coordinate system. These are supposed to represent the three axis in that system.

However, it appears that “x” seems to be used as a general placeholder name for reasons unknown, as it is the most frequently used variable name in Python.

There are also other generic variable names, which while they are full English words may have surprisingly little meaning. Variable names such as “variable”, “value”, or “string” in string variables or “number” in numeric variables actually are not as meaningful in a vacuum. If enough surrounding context exists, such as a class or function it is located in or any surrounding comments or documentation, this may be all the information that is needed. Either way, sometimes a different variable name could have been used.

3.9 Stemming

Stemming is a processing technique, which is a core functionality of natural language processing and information retrieval [43].

The most basic idea of stemming is to turn any words into their base form if they are not in their base form. It is somewhat similar, but not identical to lemmatization, which is not used in this thesis. Examples are seen on Listing 3.6.

```
words  -> word    # plural to singular base form
pasted -> paste   # past to present base form
going  -> go      # present participle to present base form
```

Listing 3.6: Examples of stemming

As seen in the examples above in Listing 3.6, at the most basic level try to remove certain suffixes in words such as -s in plurals, -d in past tense words or -ing in present participle words if one of these suffixes are present, then check in an English dictionary if the resulting word is still a valid English word [44].

Using a basic set of rules only works so far with regular cases. Irregular plural words such as “geese” which is a plural of “goose”, or “went” which is the past tense word for “go”, would not be detected by these basic set of rules. A lookup table could be used for certain special cases but with so many English words it too has its limitations.

The Porter stemming algorithm [44] would become to be the most widely used stemmer. It is implemented in Java within the Apache OpenNLP toolkit [45] and in Python within the Natural Language Toolkit (NLTK) [46].

```
<word><old suffix> -> <word><new suffix if there is one>
```

Listing 3.7: Basic idea (up to five times) of Porters Stemmer [44]

It is a simple set of rules in which certain suffixes are replaced with new suffixes or removed altogether, in five steps which cover different sets of suffixes except for the first step which handles plurals and past participles and the last step which cleans up the word.

As stemming is language-specific, stemming for German or French texts would work differently than the ones meant for English texts. Most programming language code are written in the English language, so only an English language stemmer is required.

3.10 Type hints

Type hints are annotations that were introduced in Python version 3.5 via PEP 484 [47] [48]. Python does not use any types due it being a dynamically typed programming language. The main goal of this proposal was to improve static analysis tools. This provides a mean for standardization regarding type hints and can help integrated development environments (IDEs) and other tools when it comes to code refactoring.

```
def function_name(parameter: str) -> int:
    number: int = 0
    return number
```

Listing 3.8: Example for type hints in single function which are underlined

As seen in Listing 3.8, to add a type hint to a function, a right arrow followed by the type itself has to be added between the parameters and the colon.

For type hints in variables and parameters, a colon followed by the type has to be added after the variable or parameter name itself and before the assignment operator if there is any. One special type is “None”, which is always inferred to functions which return no value, therefore it is similar to “void” in C functions.

It is also made explicitly clear that Python will always remain a dynamically typed programming language, meaning that types will never be required in Python.

Also, no actual type checking occurs during runtime in Python itself. Type hints are ignored like comments. This also means that no error occurs if the wrong type hint is written. Integrated development environments will display a warning if the type hint does not obviously match with the actual type of the variable. Special tools, such as MyPy [49] are required for type checking and were not the intended goal of type hints [47].

3.11 Type inference

Type inference is the process of inferring the possible type of a variable or the return type of a function [50]. As mentioned in the previous section, Python does not use any types.

When atomic data types are used, such as integers or strings, the type can be easily inferred by simply looking at the value that as assigned to the variable. Using Listing 3.8 as an example, it is obvious that the variable “number” is an integer. Said variable is also used as the return value of the function “function_name”, therefore the inferred type of that function is also “int”. Once the type is inferred a type hint can be added.

This becomes much more difficult if custom imported data types are used or if the variable is assigned from multiple function calls. It is also possible that the type of a variable changes multiple times in a single runtime, which is possible whenever that variable gets assigned to a different value, unlike in other statically typed programming languages.

Type inference is still an open topic of research and multiple attempts exist to solve this problem, but so far, no ideal solution exists. This will be further explored in the next chapter which introduces some approaches.

Chapter 4

Related work

The fourth chapter explains some of the related work and literature that were found during the research. It mostly focuses on type inference, an existing abbreviation expansion algorithm and some studies on variable name verbosity. One important aspect is that there were no previous attempts to improve variable names in Python so far. All related work were created by other authors and are credited as such.

4.1 Type inference

4.1.1 Type4Py

The paper “Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python” by Mir et al. [50] describes a type inference algorithm which uses a deep similarity learning-based hierarchical neural network model, unlike most previous approaches that relied on manually provided type annotations.

The abstract syntax tree is extracted from python source code to obtain the type hints required for predicting the types of parameters and the return value. No code comments are processed. After that process, various natural language processing steps as tokenization, stop world removal, and lemmatization [51] are done to that information. All uses of an argument in the function body are extracted to a sequence of tokens. The return statements are extracted too to figure out the return type. All type hints, including types from imports were collected.

For a machine learning model the extracted information is represented as vectors, which preserve semantic similarities between words. The sequences of identifiers and code contexts are concatenated into a single sequence, each.

For visible type hints, a simple vocabulary is constructed from common type hints used in Python. For any other non-standard type hint the term “other” was used.

The neural model of Type4Py uses a hierarchical neural network, which has two recurrent neural networks (RNN). Both of them are based on long short-term memory units [52] which are used to capture long-range dependencies [53]. This will capture aspects of input sequences from identifiers and code tokens. This is then summarized into two single vectors, which are found from the final hidden state of their RNN. The type prediction task is a deep similarity learning problem [54]. This allows the model to map argument and return types into “type clusters”, which are far away from other “type clusters”. It can detect user-defined types.

This paper used a Python dataset that consists of 4910 different projects. All of these projects required to use MyPy [49], which is another type checker, as a dependency. It ensures that the projects that used MyPy are much more likely to use type annotations, which are required for training the Type4Py model.

The precision of Type4Py is 64,34% for the top 1 which is higher than two competing algorithms, which have a precision of 50,85% and 45,01% respectively. For the top 3 results, the precision changes for the Type4Py algorithm to 67,72%. For the competing algorithms, it is 54,67% and 52,28% respectively. Which puts Type4Py into usable enough territory, but far from being perfect, unlike the two other algorithms.

Type4Py has an implementation, which is separated into two parts. The first part is a web server, which comes in the form of a Docker image which can be started by the user. It takes a single Python file as the input via a POST request. The response is a JSON file which contains the most likely type hint for each variable name.

The second part is a plugin for the Visual Studio Code IDE. It implements an item in the context menu when right clicking a variable name, which allows an identifier to be inferred. This will send the identifier, the function and class if applicable and the entire Python file to the server in the previous step, which infers the type of that identifier, then the user is given a choice to pick one type out of six, the most likely one is at the top of the menu and is highlighted.

4.1.2 Other type inference algorithms

Typilus by Allamanis et al. [55] instead uses a graph neural network model that attempts to infer the type with probabilistic reasoning over the code names, structure and patterns. Said network then uses deep similarity learning to learn a so called “TypeSpace”, which is a continuous relaxation of the discrete space of types. This embeds the type properties of an identifier the “TypeSpace”. It can even infer user-defined types by the use of “one-shot learning”.

Type4Py is the only algorithm besides Typilus [55] that are universally capable of inferring the type of variables. Other algorithms, such as MyPy [49] are limited to inferring arguments, return values and global constants. As the new algorithm that is explained in Chapter 5 requires that the type of the variable can be inferred. This only leaves Type4Py and Typilus as potential candidates. There also has to be an existing implementation that can be easily implemented into other programs, such as integrated development environments (IDE). Typilus does not have an implementation like Type4Py, therefore Type4Py was chosen as the type inference algorithm.

4.2 Variable name recovery

4.2.1 JSNEAT

JSNEAT from the paper “Recovering Variable Names for Minified Code with Usage Contexts” by Tran et al. [22] is an information retrieval based technique to recover full meaningful variable names from meaningless minimized single-letter names from obfuscated JavaScript code. This was intended for reverse engineering JavaScript code.

Minification is commonly used in JavaScript to reduce the size of the code file so that websites can load faster. This is also used to obfuscate the code which significantly worsens the readability of the code.

During minification, any variable names will be reduced to as few letters as possible, usually just one letter or one letter and one digit. All whitespace will be removed and any comments and documentation will be removed as well. The actual functionality of minified code remains identical to the full source code.

A basic “beautifier” [56] can restore the readability of the actual code by adding whitespace and newlines but is not able to restore any comments, documentation and most importantly, any variable names.

The approach [22] is divided into three steps, the first step is to look at the single-variable context of the variable, which is represented as a star graph. The star graph shows any assignments of the variable and shows every time when its fields are accessed or its methods are called. This is then compared to other star graphs of the entire large dataset of JavaScript code from 12,000 open source projects. The more similar the star graphs are, the better the single variable match is. The perfect match would be if both star graphs are identical.

The second step it to look at the function names and how likely a certain variable name will appear in that function name. It also looks at tokens of the function name if said name consists of multiple words.

The third step is to look at the other variables that are usually used with said variable name in the same function, named “Multiple-Variable Usage Context” in that paper. An association score is formed for a set of variable names, given how likely it is for the same set of variable names to appear in a single function.

JSNEAT has an accuracy of 69,1% with all three steps in place, meaning that it recovers about 69,1% of all meaningless single-letter variable names back to meaningful ones, which so far has the highest accuracy out of all similar approaches for JavaScript at this time of writing, having a higher recovery rate than the JSNice [57] algorithm that has an accuracy of 66,1% and the JSNaughty [58] algorithm which has an accuracy of 43%.

4.2.2 Other variable name recovery algorithms

Other approaches to recover variable names where investigated, but cannot be incorporated in Python in any usable way because how fundamentally different the programming languages are.

C and C++ are statically typed languages that are compiled into human-unreadable bytecode, which is different from Python that is dynamically typed and uses an interpreter.

The Direct Identifier Renaming Engine, short for DIRE, by Lacomis et al. [59] and the DecompIled variable Re-Typer, short for DIRTY, by Chen et al. [60], are meant to augment decompilers that decompiles any binary code back to C. These two can recover variable names with a high accuracy, up to 75.8% from all names. They must be integrated with another decompiler, in their case Hex-Rays which is proprietary. This means that these approaches are never going to be compatible with Python without very significant rework. These approaches also heavily rely on C type systems which are absent in Python.

By contrast, JSNEAT does not have any external dependencies. While it is targeting JavaScript code, Python is also dynamically typed and interpreted. Therefore it could be easily adapted to work with Python instead.

No similar algorithm exists for Python code. Python code is never minified or obfuscated unlike JavaScript code. Being an interpreted language, it means that it also does not have to be compiled into unreadable bytecode unlike C or C++, meaning that there is less motivation to “reverse engineer” Python code compared to these other programming languages.

4.3 Expansion of abbreviations

The paper “Extracting Meaning from Abbreviated Identifiers” by Lawrie et al. [21] is another important related work. It describes its own algorithm how to expand any abbreviations that are used as variable or function names to full words. It was originally used for C-type languages, but there is no reason that this could be adapted to different programming languages, such as Python.

The algorithm is split into two parts, the first part is in regards to multiple words that are written into one single word. According to the paper it can be done in two ways, either with “hard” separators such as the underscore character in snake_case style identifiers or any capital character in camelCase style, which it calls “hard words”. This type of word separation is readable enough.

However, the separation of “soft words” as called in the paper is more difficult. It uses a greedy algorithm to discover the longest prefix and the longest suffix that when added together would result into the combined soft word, the prefix, and suffix being the separated words.

The second part is the expansion of abbreviations. For a word to qualify as the abbreviation, the abbreviation must start with the same character as the full word and every character of the abbreviation must appear in the word, in the correct order.

Using the abbreviation, the algorithm looks at the relevant comments that are in the code, closely to that variable assignment which uses the abbreviation. Then it looks for the function name for any potential words that can qualify as the full word. Multi-word abbreviations are handled by looking at phrases in the comments and in other multi-word identifiers. It can also handle acronyms, where every single character in the acronym represents a word from a phrase, in that order.

Stemming is also used so that the base form of the word is always used, ignoring other forms of the word.

It also pays attention to C keywords and will not alter any function names that are from the C standard library.

From the quantitative evaluation, about 16.8% of C variable names are expanded to more meaningful words, while in C++ it is 17.8% and in Java it is only 8.7%, suggesting that Java programmers are more likely to use more verbose identifier names to begin with, according to the author, which also means that there is less of a need to expand Java variable names. Python was completely ignored in this paper. The accuracy of the algorithm, using 64 random abbreviations as the sample, was 64% if the abbreviations had three or more characters and 57% if they had one or two characters, resulting in an overall accuracy of 58%. This algorithm only returns one result to choose from to replace a name.

4.4 Studies on variable name verbosity

There are multiple user studies regarding the ideal verbosity of a variable name. For these studies there are three categories, single-letter variable names, abbreviations, and full words. These studies cannot be compared to each other directly because the exact methodology and participants are different with each study. There are four studies that are looked in detail. Two more studies exist, but they only had 9 or 6 full time professionals respectively, which is too small of a sample size. Therefore these two studies are excluded.

4.4.1 “Effective identifier names for comprehension and memory” by Lawrie et al.

The paper, named “Effective identifier names for comprehension and memory” by Lawrie et al. [61]. had 192 participants at the start of which 25% of them are students, but only 128 of them finished at least one task and only 80 of them finished all twelve tasks.

A Java applet initially starts with filling out information about the participant. After that first step, the participant had to complete 12 tasks. Each task initially shows a full Java function in which the participant could spent up to two minutes to study that code snippet. After the time elapsed, he or she had to describe the function in one sentence and rate their own confidence from 1 to 5. Describing the function was purely for distraction in order to clear the short-term memory of the participant and does not contribute towards the study. In the last step of a single task, the participant is given a multiple choice question with six possible answers. The question is which of these listed answers, which are identifier names, did appear in the previous function. One to five of these answers can be correct depending on the previous function.

There are twelve tasks because there are two independent variables, the first independent variable is the naming style of that variable name, which can either be a single-letter name, an abbreviation or a full English word. The second independent variable is the origin of the source code, it can either be based from an existing algorithm which are found in textbooks or actual production code snippets. Comments were removed from the code. The functions were then altered as necessary to create the three variants with different naming styles. As of the result, there are six permutations that cover each independent variable. This was repeated once with different code origins, to get twelve tasks. The order of the tasks were randomized to avoid order bias.

In general there was a correlation that a higher user confidence lead to more correct answers by the participants. It does depend on the code. Women, which consisted of 10% of the participants had a harder time with handling functions with single-letter variable names, though the sample size of women are small. A higher level of education and experience also correlates to better answers in this survey. Men in general were more confident in this study.

This survey concludes that using full English words as variable names are easier to remember, especially in production code snippets, but not by a large margin. It is much better than using single-letter names, but only a little better than using abbreviations which most of the time are just as good as a full word. They thought it may be a better idea to use an abbreviation over a long word with many syllables in some cases because it is easier to remember an abbreviation than a long word with many syllables.

4.4.2 “Shorter Identifier Names Take Longer to Comprehend” by Hofmeister et. al.

The study conducted by Hofmeister et al. [37] initially had 221 participants, but only 72 participants managed to finish it properly. All participants needed at least one year of job experience as a professional C# developer.

The participant, after completing a tutorial is presented with small piece of code. Only seven lines of code out of 21 can be read at any time, which is scrollable. If the participant found the error, then the participant presses the space bar and is prompted to write the line number in which the error occurred, the description of said error and a correction for that line. The time spent searching for the error is recorded, but not the time in the dialog fixing the error. The code pieces are written in C#.

The independent variables are the three naming styles, which are single-letter names, abbreviations, and single full words. Another independent variable is whenever the error is a semantic error or a syntax error. This results into six tasks that the participant has to complete. The order was partially randomized, the participants saw three tasks with semantic defects first, then the ones with syntax errors. Within each half the order regarding the naming style was randomized.

Participants could find 19% more semantic defects per minute when the identifiers used a full English word compared to the other two shorter naming styles. There is no difference in comprehension speed between these two shorter naming styles. There is no statistically significant difference in comprehension speed when it comes to syntax errors.

This survey concludes that full words should be used as names instead of single letters and abbreviations. Style guides should be updated accordingly according to this paper. It also would do a future work to do a similar survey but with eye tracking and with students rather than professionals.

4.4.3 “Descriptive compound identifier names improve source code comprehension”

The study conducted by Schankin et al. [62] initially had 182 participants that were a mix of students and professional developers, but only 88 participants managed to finish it properly within the time limit and within three attempts.

The participant is presented with a code snippet, but the participant can only see seven lines of code of 28 at any given time in a tiny window, which can be scrolled. If the participant thinks that he or she found the error, then the participant presses the space bar and is prompted to write the line number in which the error occurred and a correction for that line. An optional description can be written too. Scrolling the code is locked during that process. The time and the of the position of the visible window is recorded, especially the changes in the direction of scrolling. The code snippets are written in Java.

No single-letter variables of abbreviations were used, only simple full words or multiple English words.

There are a total of four tasks to complete, based on having two independent variables. Each task has its own code snippet. The first independent variable is the naming style of the identifier, which is either a single full English word or more descriptive multiple words which are referred as a descriptive compound variant. The second independent variable is the type of error that the participant must find and fix, which is either a syntactic error or a semantic error. The order of tasks was randomized for each participant.

For sematic defects, it was easier to find said defect when the identifiers were written as descriptive compound names, rather than shorter, single words, about 14% faster. The median value to find the defect was 210 seconds when basic words were used and 180,5 seconds when compound names were used. Also noted was how the error was found, there are less changes in reading direction with these longer descriptive compound names, which is an indication on how the code snippets are read.

For syntactic defects, there was no significant difference regardless if short words or compound words were used. How the code snippets were read while finding that syntactic defect also did not change, the results were virtually identical in the case of syntactic defects. It was also faster in general to find a syntax error when compared to a semantic defect. In the case of using short names, there were slightly more outliers than when using compound names.

The results were then split in half into two sets, one set that had the results that only consisted of results of experienced programmers while the other set did not have any results from experienced programmers. They found out that only experienced programmers could solve these after mentioned semantic errors faster, while the time difference was less pronounced for novices. As expected, experts could finish other tasks faster than novices in general, albeit not as pronounced as in the case of semantic errors with compound words.

In conclusion the paper recommended using descriptive compound identifier names over simpler identifier names, because it took less time to find semantic errors. They proposed that said descriptive compound identifier names eases the retrieval of concepts from the long-term memory in experienced programmers which uses less of the working memory to find the semantic error.

This study is similar to the one in the previous subsection, the main difference is that one independent variable is changed from single-letter variables, abbreviations, and full words to full words and multiple descriptive compound words. Only the latter one tries to differentiate between professionals and novices. A different programming language was used too, but Java and C# are also somewhat similar.

4.4.4 “Identifier length and limited programmer memory” by Binkley et al.

The study conducted by Binkley et al. [63] uses a different approach and had 158 participant, most of them which already have a degree. The experiment is substantially different. A questionnaire about the existing work experience and schooling with some personal details like age and gender had to be filled out before the actual experiment started. It is mainly about the risks of overloading the programmers short-term memory and investigating if using common terminology, which is supposedly already known by the programmer can help in memorization.

The independent variables are how long that line of code is and what identifier is replaced. The dependent variables are how long it takes to complete a single task and how correct the answer is.

The experiment which is a basic Java application initially shows a single line of code that contains a full statement. The participant can spend as much time as needed to study said line. That time spent will be recorded. Once the participant feels ready, he or she can proceed to the next question, which is actually just for distraction. The participant should type in from which kind of application this single line of code originated. The answer does not matter for this study, this is mainly to clear that line of code from the participants short-term memory. The last question actually matters again. The same line of code will be shown again, but with one identifier missing and the participant has to fill the gap correctly.

The line of code as well as the missing gap is partially randomized, with the randomization being divided into four categories: The first one has a short total line of code which is also less complex and the missing gap is expected to be memorized, such as “substring” and “length”. These identifiers are used commonly. The second category also has short identifiers but the identifiers are supposedly harder to memorize, such as “isDebugEnabled” or “getJournalEntry”. The third category is the same as the first category except that the total line of code is longer and is more complex while the fourth category is the same as the second category but also with a longer and more complex line of code. Each category has two questions, for a total of eight questions that the participant has to go through. The order of the questions are randomized. The code is written in Java.

No single-letter variables and abbreviations are used for the identifiers, the main difference are a single word and multiple words.

The conclusion of this survey is that long and more complex lines of code are harder to memorize and also take longer to process. It also shows that common and consistent identifiers are easier to memorize, especially by experienced programmers. This paper recommends to use single words which are commonly and consistently used across programmers. It highly recommends to impose limits on the length of a variable name.

4.4.5 Notes on the studies

The last two studies were done with the Java programming language, which is known for using very descriptive variable names compared to Python, which puts more emphasis on brevity. Furthermore all the studies were done in a controlled environment and the participants had no access to advanced tools or help. Therefore these results may not be directly applicable when the user uses a Python IDE.

Furthermore, the last two studies in the last two subsections seem to disagree when it comes to the conclusion. The former study by Schankin et al. [62] recommends using multiple words in variable names while the latter study by Binkley et al. [63] does not recommend using multiple words and rather recommends using a single full English word to describe a variable. Neither of these two investigated even shorter names, such as abbreviations or even single-letter variables. This is due to different methodology.

Based on the previous studies, this thesis will go ahead and expand names to single English words, not multiple ones. The other reason is that this is done for the Python language, which encourages brevity. Long descriptive names feel more out of place in this programming language than in Java.

4.5 Other related work

There is paper named “A large-scale comparative analysis of Coding Standard conformance in Open-Source Data Science projects” by Simmons et al. [9]. This investigates if data science projects on GitHub follow PEP 8 [24] code conventions. It compared 1048 data science projects from the Boa data science dataset [64] with 1099 non data science projects that have a similar level of code quality based on the number of GitHub stars. The results are that data science projects have a higher number of functions that use too many parameters and too many local variables. It also proves that data science projects do indeed have different naming conventions, but other than that the differences are not that large.

Some of its methodology will be used in Chapter 7 for the evaluation of the algorithms in the next chapter.

The Boa Data science dataset by Biswas et al. [64] is a dataset that consists of 1558 data science related GitHub repositories. The main criteria are that is uses one data science related GitHub topic and uses one data science library. Finally, it had to have at least 80 GitHub stars. It was also used in the paper mentioned above. More details on this dataset are in section 7.3, as this dataset will be used in the upcoming evaluation in chapter 7.

Chapter 5

Concept

This chapter explains the theoretical concepts of both abbreviation expansion and single-letter variable replacement which will be used as a solution to improve existing variable names. This will also create the resources that are necessary for the IDE plugin and for the later evaluation.

This will use the related work of Lawrie et al. [21] as the basis for the first approach and the related work of Tran et al. [22] as the basis for the second approach. Type inference by Mir et al. [50] will also be used for abbreviation expansion.

5.1 Architecture for data model creation

The program mainly consists of four parts. The first part are the prerequisite resources such as the generic Python code dataset and the English dictionary. The second part is the abbreviation expansion algorithm. The third part is the single-letter variable replacer. The fourth part which is described in Chapter 6 is the technical implementation of the plugin. One rationale is that the initial creation of the resources will be slow, but only needs to run once, but the actual process of replacing the variable name will be near instantaneous.

5.1.1 Generic Python code dataset and other resources

The first part is to extract the generic Python dataset by cloning 25557 GitHub repositories and then removing any repositories that area already in the Boa data science dataset and applying additional criteria such as having Python 3 compatible code files. This reduces the number of GitHub repositories from 25557 to 20925. The exact criteria are described in section 5.3.

Before any of that, there is a brief explanation of the English dictionary which is explained in the section 5.2.

5.1.2 Abbreviation expansion

The second part is the “Abbreviation expansion”. The basic idea is derived from a previous work by Lawrie et al. [37]. Said work was only applied in any local C/C++ code, but this time around a much larger dataset of Python code is used, which was obtained from GitHub and filtered. The program looks at every single line of code for variable names that can be expanded if there are corresponding full name tokens on the other side of the variable assignment. The result is an initial tree data structure that maps abbreviations to full words. The first pass is described between the sections 5.5 to 5.15.

This has to be repeated, the second round also takes the initial abbreviation lookup tree into account to resolve abbreviations in multi-word variable names which could not be resolved in the first round due to a lack of said tree in the first round. Full English words will be prioritized. The recursive passes are described from section 5.16 to section 5.20. This creates the final abbreviation lookup tree data structure. The actual abbreviation expansion, once the necessary resources are created, is described in section 5.21.

As a side effect of the abbreviation lookup tree data structure it is possible to use that to shorten excessively long variable names by applying it in reverse. This is described in section 5.22.

5.1.3 Single-letter variable replacer

The third part is the “single-letter variable replacer”, which mostly uses the work by Tran et al. [22]. The idea is to treat these meaningless single-letter variable names as unknown tokens whose full variable name must be recovered.

It attempts to look at the interactions of each variable, such as the (initial) variable assignment and when said variable is directly used or one of its fields or methods are used. The function it is located in is also factored in. These interactions are then noted into a so called variable relation list.

To speed up access times, these lists are put into a decision tree which is sorted by the size of the variable relation list, then by the function name. It can then be exported to the hard drive. For frequently used lists, these can be put in a cache instead which is loaded into system memory. These are described in section 5.23 to 5.28.

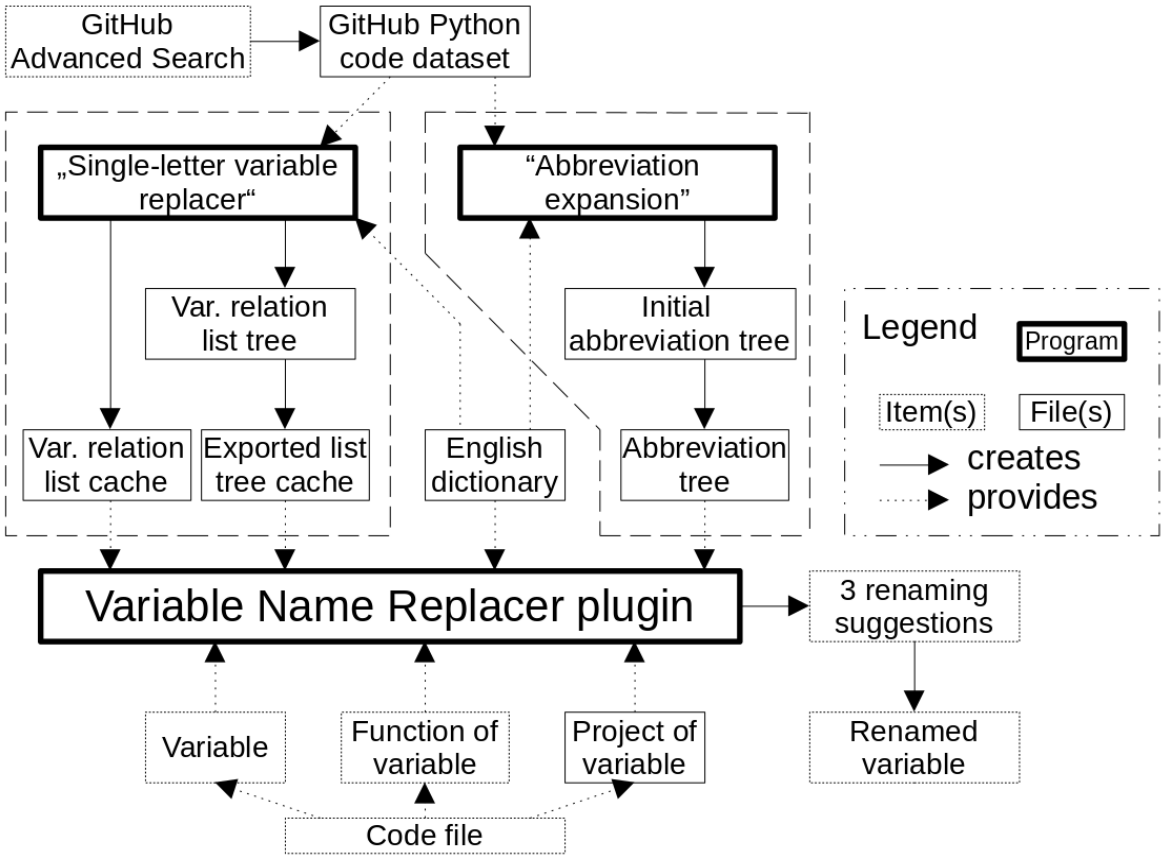


Figure 5.1: Basic architecture for data model creation

The actual process of finding a replacement for the single-letter variable, which involves creating a local variable relation list and then attempting to match it with one of every of the variable relation lists is described in section 5.29. A graphical depiction of the architecture can be seen in Figure 5.1.

5.1.4 Other features

Some removed features are explained in the last section 5.30.

The “Variable name replacer plugin” is the actual end-user implementation that require the results from the previous parts. That will be explained in Chapter 6.

5.2 English dictionary file

An English dictionary is required for the stemming and information retrieval algorithms that are explained in the later sections. A simple list of English words is sufficient enough. Said list would be stored as a simple text file. Every word is stored as a single string line, which are separated by newlines.

There is a list of English words that can be used, which was originally created by Church [65]. These are the 100000 most frequently used English words on Wiktionary. However, it was compiled in August 2005 and its last pull date was in January 2012, meaning that it is probably outdated. It also has 28931 duplicates and other similar words that only differ in capitalization, so there are about 71069 words. While natural English words should not change that quickly, very specific words may be missing. It also only contains the base form of the words. The words are sorted by the popularity, meaning the most frequently used words are at the start of the list.

During early tests the dictionary was missing many English words. They had to be manually added to the dictionary by finding the most frequently used non-English variable names, then manually inspecting them and adding these English words into the text file, manually. Commonly used abbreviations and acronyms are not added as they are not English words by definition.

A few Python keywords, which are required by the PEP 8 convention [24], such as “self” and “cls” are manually added to this dictionary as well.

On the other side, there are some non-English words in this text file, which had to be these words are in string length, then these files can be manually inspected.

The algorithms in the later sections generally treat all words that have one or two characters as non-English words, so they are ignored entirely. One reason is that Pylint treats any non-whitelisted name that is one or two characters in length as non compliant.

Because of these changes, the size of the dictionary changed from 100000 words to 99256 words.

Given that the language can change over the time, especially in the fast-living world of technology, this dictionary will become outdated at some point as new technology terminology and slang will be invented over the time. This dictionary is not capable of updating itself and requires manual review at some point. The text file can be easily edited to add or remove words if necessary.

5.3 Generic Python code dataset

A large Python code dataset is required for these information retrieval based algorithms that are described in the later sections. Some of these repositories use libraries that are typically used in data science projects, but these can be used for other reasons that are not related to data science [9]. They will therefore not be excluded to get as many repositories as possible. Any repositories that are also in the data science dataset are excluded to avoid duplicates.

5.3.1 GitHub

GitHub [66] is a development platform which hosts more than 200 million Git repositories made by more than 73 million developers. It claims to have more repositories than any other competitors. For this reason the code dataset will only be retrieved from GitHub. The other reason is that some repositories are mirrored on other platforms, therefore avoiding duplicates.

5.3.2 GitHub Search API

To find the repositories, the GitHub Search API was used [67]. The main criteria for these repositories in the search are that they used actual Python code and that the date of the creation of the repository is not older than January 1 2016 to increase the chances of not finding older Python 2 code.

Initially all repositories also must have at least 100 GitHub stars, which is currently the only way for users to rate GitHub repositories positively, similar to a Like button. Higher star count therefore implies a higher overall quality. It can also be used to limit the number of found repositories. As of February 15 2022, 27,302 Python repositories that have at least 100 GitHub stars. The API call is seen on Listing 6.2.

```
url = new URL("https://api.github.com/search/" +
      "repositories?q=created:>2016-01-01+stars:"+ MIN_STARS +".."+ MAX_STARS +
      "+language:python&sort=stars&order=desc&per_page=100&page="+page);
```

Listing 5.2: URL of advanced GitHub search

The GitHub Search API does have a limitation that it only returns up to 1000 results per query, which are divided into ten pages of results that contain 100 results each.

To get around this limitation, the range of stars are specified per search. Each search has a minimum and maximum of stars. Using this range limit, it is possible to get under 1000 results with a set of ten search queries which can find up to 100 results each for a total of up to 1000 results. After one set of ten search queries are completed, one for each page, the number of minimum and maximum stars are then incremented as described in the formulas below for the next set of ten search queries.

$$newMinStars = oldMaxStars + 1$$

Listing 5.3: New minimum GitHub stars range

Increasing the minimum range of stars is straightforward as it only needs to be the same number of the old maximum range of stars plus one as seen on Listing 5.3. The next Listing 5.4 is seen on the next page.

$$newMaxStars = oldMaxStars + oldMaxStars - oldMinStars + 1 - a$$

Listing 5.4: New maximum GitHub star range

Increasing the maximum range of stars is slightly more complicated, as an addition is required from the old maximum range of stars plus the old range of stars plus one. The variable “a” depends if the below condition is met, if yes, then “a” is 1, otherwise it is 0.

$$numberOfFoundRepoLinks < \frac{oldMaxStars - oldMinStars}{oldMaxStars - oldMinStars + 2} * 1000$$

Listing 5.5: Condition to increase range of GitHub stars

This condition in Listing 5.5 determines if the range has to be made larger. If the number of repository links found in the search queries is below the threshold given from the right hand side equation, then the range is increased. 1000 refers to the maximum number of repositories that can be obtained from one request.

For an example given the minimum range of 118 stars and the maximum range of 120 stars, it would return 472 GitHub repository links, which is less than the number of 500 given from the right hand side of the equation. Therefore the next set of search queries will have a larger star range by one. For that matter, said search queries would return 665 repository links, which is higher than the threshold of 600, therefore the star range will not be larger in the next set of search queries.

The reason why the range of stars has to be increased is because there are fewer repositories the more GitHub stars it has given an exact number. For instance, there are only 21 repositories that have 399 stars. Figure 5.6 illustrates this for Python repositories.

This will be repeated until zero repositories are found. The entire process was done by hand and was not automated.

A rate limiter is required to due to API rate limits set by GitHub [67]. Only 10 unauthenticated requests per minute are allowed, while 30 authenticated requests per minute are allowed if user credentials are provided.

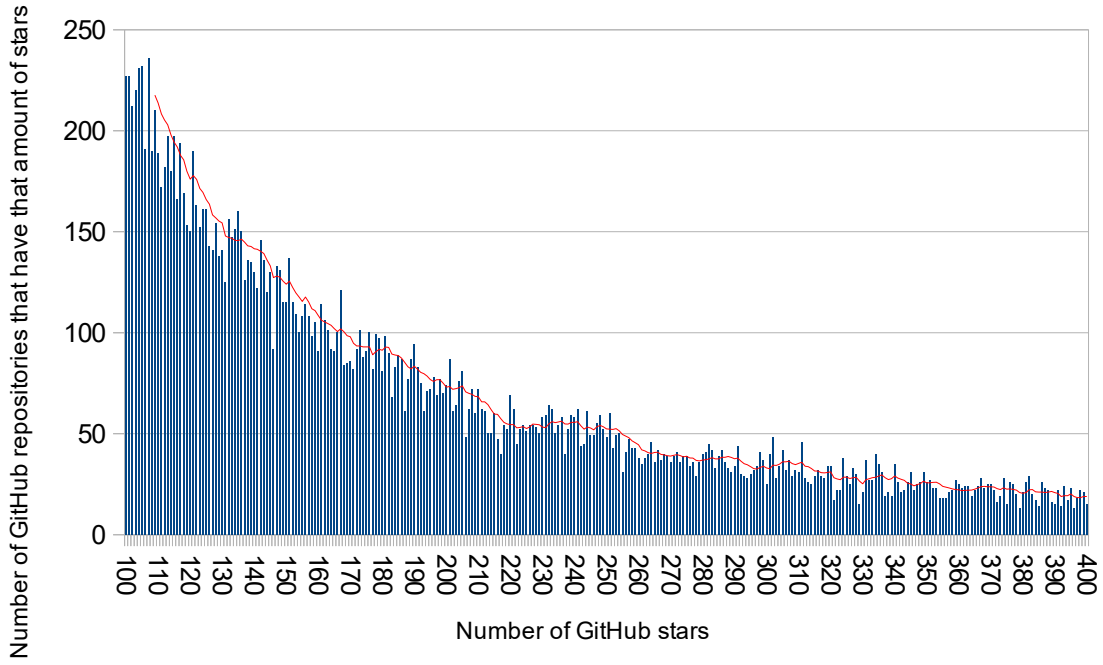


Figure 5.6: Number of repositories with that number of stars. Red line is the average

5.3.3 Git cloning script generation

One search query from the previous section will contain multiple links to any GitHub repositories. The search query itself is a HTTP GET request which returns a JSON document as a single large line. That document will be parsed to look for the key "full_name" which has the value that contains both the author of the repository and the repository name that belongs to that author. One example is seen in Listing 5.7. This key-value pair should be found 100 times if 100 results are found from the search query.

```
"full_name":"author/exampleRepo"
```

Listing 5.7: Example key-value pair

The value contains both the author, which is behind the slash character and the repository name which is after the slash character in the value string. Using both the author and the repository name, a single line for the cloning script can be generated. The example in Listing 5.7 is used to create the example in Listing 5.8.

```
mkdir author;
cd author || exit;
git clone --depth=1
https://00000000:00000000@github.com/author/exampleRepo;
cd ..;
```

Listing 5.8: Example line from GitHub cloning script, split up for readability

Listing 5.8 describes a single line from the GitHub cloning script. The script itself has to be placed in the folder of the code dataset.

The first step of the script is to create a directory for the author of the repository. If the directory already exists, which is possible when one author owns multiple repositories, then nothing happens in that step.

The second step is to change to the new author directory. In the very unlikely case when the directory does not exist, such as when the hard drive is full, then the script exits.

The third step is to perform the actual cloning of the repository. A shallow clone occurs which only clones the most recent state of the repository, indicated by --depth=1.
This process will always create a new directory named after the repository. Said directory contains the actual repository. Meaningless credentials were inserted in case they are prompted. In that case, that one repository will not be cloned and it is fully automated.

The last step is to simply change to the parent directory. These four steps will be repeated until all lines in the cloning scripts are used up.

One unlikely case which may occur when using an outdated cloning script is that there will be a prompt for a GitHub user name and password. What actually happened is that either the repository in question has been deleted or that it has been marked as a private repository by the author since then. If the prompt occurs, these credentials will be used which results in a failure, not cloning that non-existent repository. This should not occur if the cloning script has been generated very recently, as all repositories in the search do exist and are public.

As mentioned previously, there must be enough hard drive space to save these repositories. At least 2 TB of hard drive are recommend, certain tricks can help to reduce the space that it uses up, more on that in the subsection 5.3.5.

5.3.4 Filtering the Python code dataset

After cloning all repositories, they have to go through another filtering process. The generic python code dataset will be filtered first as seen on Figure 5.9.

The first step was to remove seven excessively large repositories that simply take up too much hard drive space. These repositories also have very few actual Python files.

The second step is to remove 165 repositories that contain malicious code [68] or other malware. This is done with a free anti-virus scanner. The reason is that these files would be examined later and the risk that it might activate any malware is too great for this thesis.

The third step is to remove 103 repositories for not having any Python files. Python files always end with a .py file extension. Why they are considered to be Python repositories in GitHub is still unknown, some of them have .ipynb files which are actually meant for the Jupyter Notebook that contain Python code but are not actual Python files.

The fourth step is to remove any repositories that are also found in the other Boa Data Science dataset [64]. This is to avoid any potential duplicates.

The fifth step is to remove any repositories that are not compatible with Python 3. Python 2 has been discontinued since January 1 2020 [69]. Pylint does not support Python 2 either. Python 2 code has certain features that are absent in Python 3, such as a different print statement, the xrange() function has been replaced by range(), integer division is handled differently and more [70]. To detect Python 2 code, use both the Python 3 interpreter on the code to check if certain errors occur that would not occur if the Python 2 interpreter is used. If this is the case then it is Python 2 code and the corresponding repository has to be discarded. 3331 repositories were removed for that reason

The sixth step is to remove any repositories that cause problems with Pylint [71], such as memory leaks in Pylint or never completing the process. 38 repositories were removed.

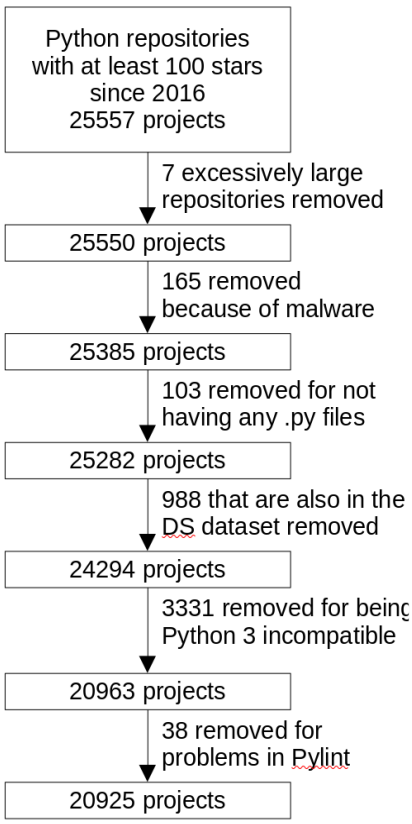


Figure 5.9: Filtering the Python code corpus

The number of removed or private repositories may increase in the future if the same repositories are cloned again in the future. Also, all these steps were done manually.

All Python code files that have more than 10000 lines are ignored, because these are typically not written by humans, but are automatically generated. This is to avoid huge files that would be outliers in the later evaluation. This also applies to code files with a single huge line of code that is longer than 1000 characters in a single line.

5.3.5 Saving space on the hard drive

As mentioned above, some repositories take up too much hard drive space and had to be removed. There are some other files that are not relevant for the later operations and can be removed safely for the purposes of this thesis.

Every locally cloned Git repository contains a .pack file located in the .git/objects/pack directory in each local repository folder. It can be removed, though Git operations may no longer work properly. In fact, any non-Python file could be removed, as only the actual Python code will be investigated further and the code itself does not required to actually function properly as these files will not be interpreted at any later step.

One must be careful with deleting files to not accidentally delete the wrong files. The deletion process itself can be quite intensive with potentially millions of unnecessary files being deleted. It is therefore recommend to perform any of these tasks on a solid state drive (SSD) to reduce the time required.

5.3.6 Reconstruction of the cloning script

To reconstruct the cloning script from the Python code dataset the first and second level of subdirectories from the initial directory of the Python code dataset will be scanned. The first level use the names of each author, the second level are the repository names belonging to said author. Using both, a new GitHub cloning script can be constructed similar to the seen in subsection 5.2.3.

5.4 Other prerequisite notes

Some repositories include a virtual environment (venv). These will be completely ignored as any code inside these virtual environments were not initially created by the authors themselves. A basic stemming algorithm is required to find the base from of each word, but a lemmatizer will not be required.

5.5 Overview of abbreviation expansion

Sections 5.6 to 5.15 describe how to obtain the full words that correspond to the related abbreviation from a Python statement. The rationale is to take an existing variable name, which abbreviated and expand it to a full English word, using the existing characters of the abbreviation and any additional context derived from the code, which should be easier to understand by a human, especially to newcomers who are not familiar with the terminology and abbreviations used by experienced developers. This also tries to retain some the original meaning as much as possible.

The Python files need to be preprocessed first, then every line will be read. If the line is a Python statement which has an assignment operators, then that statement will be split into the left and right sides. This is detailed in section 5.7.

The left side is processed, which extracts the variable name(s) and possibly a type hint. It is described from section 5.8 to section 5.13.

The right side is processed to extract potential full English words, which is described in the section 5.14 and section 5.15. Then if a mapping is found it is stored in the abbreviation lookup tree data structure. Then check if the variable name is an abbreviation of the full word. If yes, then it is a correct mapping which is added to the tree.

Not shown on Figure 5.10 is the second pass in the abbreviation expander, the sorting and cleanup of the abbreviation lookup tree data structure. These are described in the sections 5.16 through 5.20. Finally, there is the actual step of expanding an abbreviation in a local code file. This is not depicted in Figure 5.10, which is described in section 5.21. Unless noted, all steps except for the actual abbreviation expansion on the local code are done with the GitHub Python data corpus.

The heuristic are all the letters in the variable name, in the correct order, in order to get the full word.

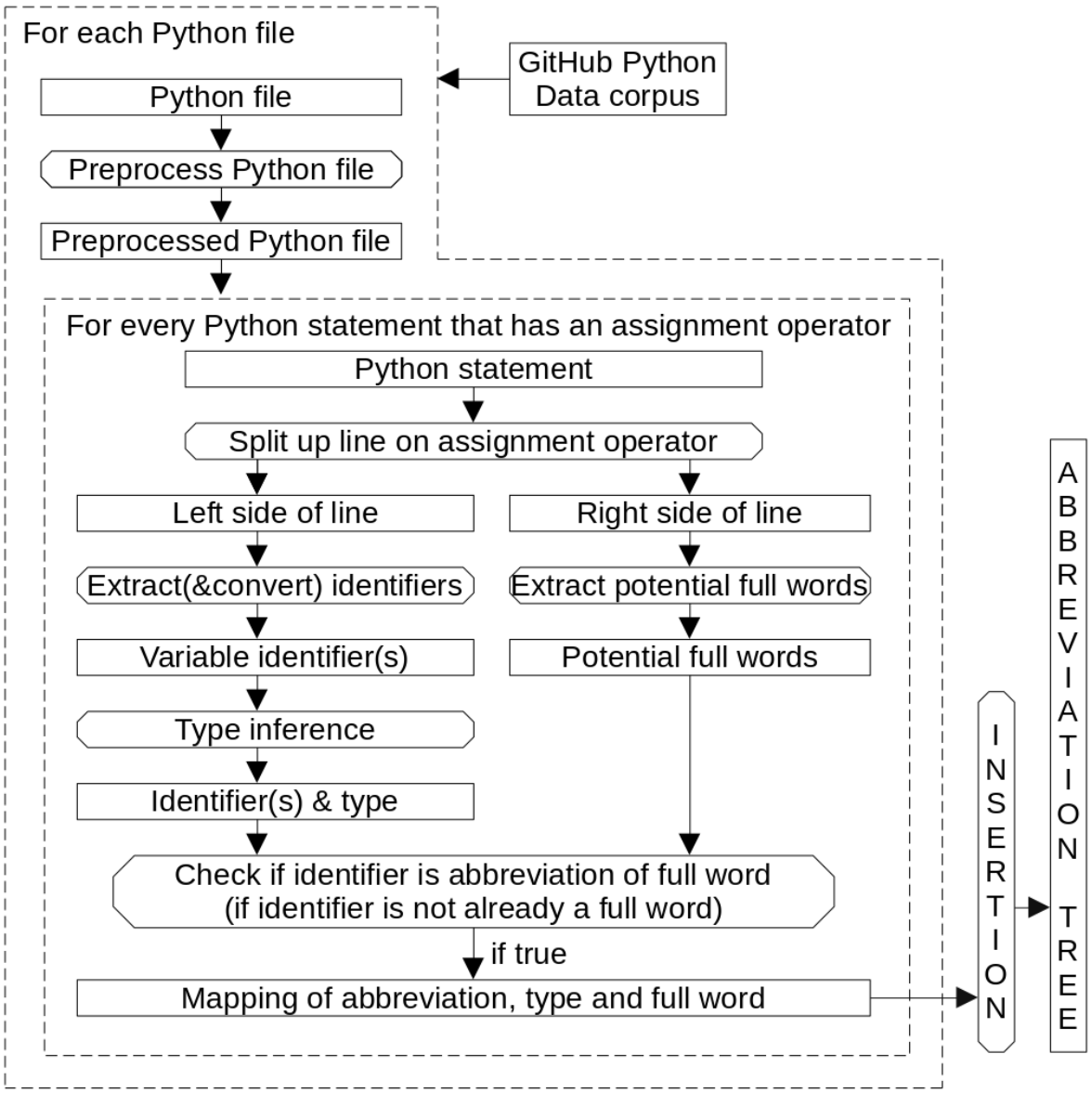


Figure 5.10: Simplified single pass of abbreviation expansion

5.6 Abbreviation lookup tree data structure

Before explaining the process of expanding abbreviations the data structure of storing the abbreviations as key with the full words as values has to be explained in detail because of a few additional details. A tree data structure was used to store the information.

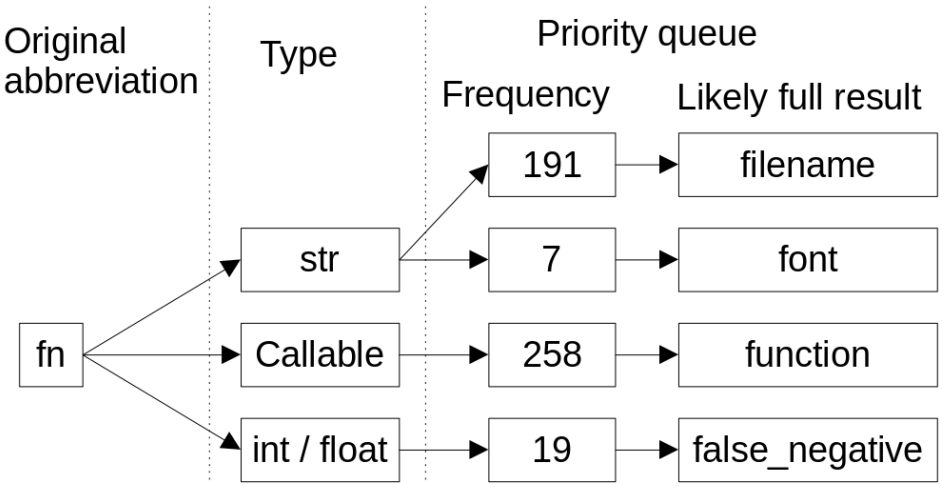


Figure 5.11: Example of abbreviation lookup tree data structure

The first level in the tree is the abbreviation itself, which can have several nodes. Said abbreviation must have at least two characters or else it is not an abbreviation.

The second level is the associated type that was inferred by type inference. This is usually always given but the type inference itself is not perfect. In rare cases when the type cannot be inferred the type “Any” is used.

The third level is a priority queue, for every parent node. The priority is based off the frequency of how often the likely full result occurs. The priority can be increased if the likely full result consists of full English words. This queue can contain multiple items. In the event that the function name level is ignored, all priority queues that share the same first and second level nodes are combined into a single large priority queue. The priority can be thought of the popularity of the likely result.

Internally the lookup tree structure is implemented as three nested maps. The last map uses a priority queue as value. A function exists to get multiple priority ques that share the same first and second node. No duplicate entries are allowed.

A tree structure was chosen to improve the time performance from the end-user client perspective. Ideally when all nodes are used, then only three steps are required to get the correct priority queue, basically giving it an ideal time complexity of O(3). On average, given the chance of a non-matching function name, the average time complexity is closer to O(n). In the worst case when the type name is missing, the worst case time complexity is O(n*n). Since no duplicates are stored, the space complexity should be O(n).

One example of a small part of the abbreviation tree data structure is seen in Figure 5.11.

5.7 Preprocessing the code

Before the code dataset is processed, it needs to be preprocessed. It involves two steps, the first one is the removal of string contents, comments and documentation and the second one is turning multi-line statements into a single line statement. All operations occur inside the memory, meaning that the actual code stored on the hard drive will not be altered.

5.7.1 Removal of string contents, comments and documentation

The contents of any strings, comments and documentation are not relevant for the later operations and can even cause confusion during the later processing, as docstrings were read instead actual code [72]. As a result, these will be removed.

To remove single Python comments which are indicated by a single number sign "#", simply remove anything after that character include the character itself.

Strings are indicated either by an opening and closing single-quote (') or double-quote (") characters. Any content inside these quotes will be removed, but not the actual quotes to keep the code intact, but remove any quote characters that are escaped with the backslash character.

Docstrings are indicated by three opening and closing single-quote or double-quote characters. The logic of removing them is the same as with strings.

5.7.2 Converting multi-line statements to a single-line statement

Some statements in Python code are split up into multiple lines to conform to the 79-character limit set by PEP 8 [24]. Said code convention also sets up how to split up any long lines, there are two possibilities to to that.

The first and preferable approach is to not use any closing brackets in a single line to split a long line. Python does not use semicolons to end a statement. Instead, it checks whenever the number of opening and closing brackets ("[", parentheses ("(") and braces ("{") are equal in a single statement. The last line of a multi-line statement is the entire line with the last closing bracket, including anything after the bracket in that line. One example how not to do a multi-line statement is seen in Listing 5.12.

```
variable = 1 + (2     # start of statement
       + 3 ) + 4 + 5  # part of previous statement
       + 6      # not part of the previous statement, error!
```

Listing 5.12: Example multi-line statements

To turn it into a single line, simply count the number of opening and closing brackets, braces and parentheses over multiple lines until that number is equal and combine any multiple lines into one. Excess whitespace and newline characters are to be removed.

The second and less recommended approach by PEP 8 [24] is to use a backslash ("\") at the end of the line. This should be avoided whenever possible. The process is of combining lines in that case is to check for any backslash characters at the end of the line, remove the backslash character and combine the next line with the line that used to contain the backslash character.

All later sections use the generic Python code dataset that was described in the previous sections.

5.8 Extracting variable names from a line

To extract variable names from the line, look at lines that do not start with the Python keywords “def”, “while”, “for”, “if”, “elif”, “else”, “return”, “try”, and “except” [73]. Then the line must contain a Python assignment operator, such as “=”, “+=”, “-=”, “*=”, or “/=” [74]. If it has one, take the left side of the equation.

Any brackets that indicate list use are not part of the variable name and are removed. Type hints, which do indicate the most likely type are also not part of the variable name and are removed from the name. If class members are used, then only the member name is considered. It is the name after the last dot character is there is one present. Anything before that is removed. Any list notations are removed as well. This can be seen in Listing 5.13. Sometimes multiple variables are declared in a single line, in which case these variables are separated by commas. All of them will be extracted.

```
class_name.var_name[2]: list[int] = [0]
         # ^^^^^^^^ only extract this potion
```

Listing 5.13: Example for extracting variable name from statement

Check if the variable name is already a full English word by checking if it can be found in an English dictionary. If this is the case then that variable name does not need any further processing and will be discarded. Otherwise that variable name will be processed further in the next seven sections.

5.9 Normalizing to snake_case

To conform the previous naming conventions regarding variable names as in PEP 8 [24], all variable names must be converted to snake_case if that is not the case. There are two specific cases and one non-trivial case which will be discussed later in section 5.10. To separate the words in a variable name, either underscores or capitalized letters are used. This follows the standard rule that all variable names must only contain alphanumeric characters or the underscore character. Said names must never start with a number. This does not handle two combined words written as one, this is handled in the next section.

5.8.1 From camelCase to snake_case

If the variable name uses camelCase [75], where the words are separated by uppercase characters, then an underscore character is added before the uppercase character if the character before the capital letter was a lowercase character. After that all uppercase characters will be converted to lowercase characters. For example, the word camelCase is converted to camel_case.

5.8.2 From all uppercase to snake_case

If the variable name consists entirely of uppercase letters and the words are already separated by underscores then simply turn all letters into lowercase letters. This can also be done in reverse if the variable name is actually a constant that needs to be capitalized.

5.10 Splitting up two words that are written as one

Unlike in the word splitting algorithm used by Lawrie et al. [21] which uses a greedy approach, this one takes advantage of the dictionary that was sorted by popularity.

One combined word, which can be either found alone or as part of another properly separated multi-word name, must have a length of at least six characters, based on the assumption that a proper English noun has at least three characters. Some English verbs only have a length of two characters but including two letter words increases the chance to cause wrong splits. It should be possible to realize this with three separate words, but it gets more difficult with every additional word because each popularity has to be taken into account. As such, fewer words are preferred.

The initial split index is set to three. Two substrings are formed from the combined word, the first one starting from index zero, inclusive, to the split index, exclusive, while the second one starting from the split index.

Then it is checked if both substrings are full English words that are found in the dictionary. If this is true, then fetch the popularity rank from the dictionary from both words. A lower popularity rank (ex. 1st item is more popular than the 2nd item) indicates a higher popularity. If the sum of both popularity numbers is lower than the existing sum, the better, in which case that split is then assumed to be the best split.

The split index is incremented and the above step is repeated, until the string index equal to the length of the string minus three is reached. The best result is then taken. This currently only supports splitting words into two, not three.

5.11 Alternative reversible stemming algorithm

This section is attributed to the Porter stemming algorithm [44] as it is based on it. However, that stemming algorithm is not reversible. Recovering a stemmed word to the original state is not possible without the information of the original state, as one stemmed word could be derived into many other words [77].

Abbreviations that are plurals usually end in -s or in -es, which is an indication that it is a plural. Using this hint, that suffix is removed and the abbreviation is marked as a plural.

Once a full word is found for the abbreviation, the last characters of the word are checked. If the last character ends with -o, -s, -x, -sh or -ch, then the suffix -es is appended instead of the suffix -s to ensure that the plural spelling remains correct. If the word ends with -y, then the -y suffix is removed and replaced with the -ies suffix [78].

Any numeric suffixes are temporarily removed in the same fashion how plurals are handled. This also applies to a suffix which starts with a number followed a single alphabetic character, such as the suffix “2D” or “3D”. Once the full word is resolved these suffixes are added back in.

The reason is that the singular base form of the word is more likely to have a resolved full word than its plural counterpart. This can also be done in reverse in certain rare cases where the plural form can be resolved into a full English word, but not the singular base form. One example is “hparams” which normally can only be resolved by its plural form. The supposed full word is “hyperparameters”.

5.12 Obtaining the function name of the variable

All function names are found after the Python keywords “def” [73] or “async def” [76], which indicates the start of a Python function.

The variable name that belongs to the function must have at least one more leading whitespace character than the function declaration. The variable declarations line number must also be greater than the line number of the function declaration by at least one. After that, from the line of the variable declaration, the closest function declaration will the searched, based on both criteria above. If the variable shadowing occurs inside an inner function then the inner function will be selected.

The function name, which is after the keyword “def” and before the parentheses which are the parameters is picked.

Function names are no longer used in the abbreviation lookup tree data structure to categorize words because then tree would be too granular, which is prone to overfitting, but are still required for type inference which is described below.

5.13 Obtaining the type

Using the Type4Py type inference system by Mir et al. [50], it is possible to infer the type of a variable. Its Docker instance has to be started first once. Currently it is treated as a black box. The Python code file will be used as input for this system. The output of that system is a JSON file that contains the most likely type for a given variable that belong to that function. To get the type, both the function name and the variable name is required to locate the correct variable inside the JSON output file. From there on, the type is located and this will be used as the type.

In the unlikely event that no type could be inferred or if there is an I/O error at the system, then a “null” type is returned and the from there on type is treated and stored as “null”. Said I/O errors can occur if the Python code has syntax errors.

See section 4.1 for further details on the Type4Py type inference.

5.14 Tokenization of statements

The right side of the statement needs to be tokenized for further processing. The left side is already handled in section 5.8, so it will be ignored. Tokenization is the process of of splitting a sentence into its components, the individual words are then called tokens [79].

All non-alphanumeric in the ASCII namespace, except for the underscore character are considered to be separators for the Java Scanner, based on the rule that variable names can only have alphanumeric characters and underscore characters. The Java Scanner class itself it then used to separate the tokens. One example is seen below.

```
fn = function_code(perks_list=[], return_val='')
     ^^^^^^^^v^^^^ ^^^^^v^^^^     ^^^^^^v^^^
     Valid tokens are marked as seen above
```

Listing 5.14: Example tokenization

Certain keywords such as “and”, “or”, and “not” are also excluded if they are separate words because these are boolean operators. The same applies for the words “True” and “False” for the same reason. The word “None” is also excluded.

Once the tokens are obtained, these tokens will be converted to all lowercase snake_case if they were using camelCase [75]. This was already described in section 5.8.

From there on, subtokenization of these tokens is possible, also using the Java Scanner in the same way as in the tokenization but using the underscore character as separator. This allows the individual extraction of single words if necessary.

Using the same example from Listing 5.14, the subtokens of “function_code” are “function” and “code”. Some abbreviations do use multiple words though, so both the full token and the individual subtokens are considered.

5.15 Extracting full word from abbreviation and line

It is now possible to find out any potential full words for the abbreviation.
Every single line in the Python code dataset will be scanned. If the full token consists of multiple subtokens, then these subtokens will be looked at first.

As described in the algorithm by Lawrie et al. [21], every subtoken and token must start with the same character as the abbreviation. If this is the case, then check first if the potential full word first characters match the abbreviation. If that is true then this full word will be returned. If that is false, then check if every character in the abbreviation occurs in the full word, in the correct ordering.

This means that, from left to right, every character in the potential full word will be looked at and must match with the most recent character that has yet to be matched, starting from the second character, as the first character was already matched with the first character from the word. Only if the second character is matched, then the third character will be searched, from the same character index where the second character was matched and so on. If all characters from the abbreviation are matched, then the word is considered to be the full word of the abbreviation. The caveat in both cases is that the tokens must never exactly match the abbreviations.

In the case nothing is found from the subtokens, the full token will be looked at. One specific case that can only occur in multi-word names are so called acronyms where every character in the acronym represents the first character of every subtoken, in that order [21]. If the number of subtokens are identical with the length of the abbreviation, then take the first characters from each subtoken, in that order, and check if these are identical with the abbreviation. If this is true, then it is an acronym and it will be returned as such.

Otherwise the same process as used in the subtokens are applied to the full token itself, with the same returned results if applicable. If there is no match at all, then neither the token nor the subtokens are the full words of the abbreviations and nothing will be returned. Before it can be stored both function name and type has to be discovered which will be handled in the next sections.

The abbreviation, type and its resolved full word are stored in the abbreviation lookup tree data structure. If no entry exists, this entry will be created with an initial priority of one. If this entry will be added again, then the priority is increased by one. A higher priority indicates a higher popularity and also it is also more likely that the resolved word is a proper English word as this is explained in section 5.17.

Using the example from listing 5.14, the abbreviation “fn” would resolve into “function”, since there is already a subtoken “function” that meets the criteria, since it starts with “f” and contains one letter “n” after the letter “f”.

5.16 Recursive resolving of abbreviations

In some cases it is possible that the resolved word of an abbreviation is another slightly more detailed abbreviation, which can also be found on its own on the abbreviation lookup tree data structure. One example is seen on Listing 5.15.

```
fn -> func -> funct -> function
```

Listing 5.15: Exaggerated example of recursive resolving of abbreviations

With the known abbreviation lookup tree data structure, look at every leaf node. If the “full word” at the leaf node is actually an abbreviation, which is found in the newly created tree, then it will be replaced by the “full word” of that abbreviation. Given that there is chance that it could be another abbreviation, this process is recursively repeated until either the end result is a proper full English word or if the abbreviation cannot be resolved any further.

Given the rules of how resolving abbreviations work as in section 5.13, every subsequent result will have a longer string length than the previous one and said result will always contain all characters from the previous result.

Only the most likely result is chosen to avoid too many possible results. If a single abbreviation is resolved into three slightly more meaningful abbreviations, then these three more meaningful abbreviations could resolve into a total of nine even more meaningful abbreviations, and so on. In other words, exponential growth. In contrast, if only one result is chosen for recursively resolving an abbreviation then it will only result into one result.

Once all abbreviations in the abbreviation lookup tree data structure are recursively resolved, the results will be updated as such.

5.17 Priority and English words

Resolving abbreviations may yield different results depending on the abbreviation. Ideally, the result is a full English word or multiple English words, but the result can also be non-English words, even after the abbreviations are recursively resolved as in the previous section.

Therefore, results containing full English words will be prioritized. If the result does contain a full English word as found in the dictionary, then the priority will be multiplied by 1000. This ensures that it is far more likely to be the top result.

If the result contains multiple tokens separated by the underscore character, then all tokens must be full English words to qualify as a full English word in this section. If a single token is not a full English word while the other tokens are, then the multiplier is reduced to 10. If two or more tokens are not English, then it is considered for this purpose to be not an English word and the priority value will remain the same.

Of a separate but unrelated note are the difference of English words which are nouns and the ones that are not. Detecting nouns and things that are not nouns is possible using a Part-of-Speech tagger [80], which is used by lemmatizers. However, these taggers require full English sentences in order to work, meaning that these taggers do not function properly in single words or when multiple words are combined into a single word. For that reason, such a tagger is not implemented.

5.18 Multiple abbreviations in a variable name

The same approach already described in section 5.15 can be applied to multi-word abbreviations. The entire abbreviation may not have been resolved in the first pass or in the second pass, but it is possible to resolve its individual words to full words. The same main difficulty applies here as well, but now over multiple words.

Unlike in section 5.16, some variable names that contain multiple abbreviations could never be resolved at all until this point. They can be still found on the list of non-English variable identifiers. Every identifier in that list will be checked if they contain multiple tokens that can be resolved into full English words. If this is the case, then the identifier is resolved, using the most likely result for each token, and then placed in the abbreviation lookup tree data structure.

Only the most likely result is chosen to avoid too many possible results. It would be 3^n results if resolving a single abbreviation yields three different answers, n being the number of tokens in a single identifier. An identifier with four tokens that can resolve into three different full words each will cause 81 different results. Therefore only the best result is taken for every token, only resulting into one single result.

The type is ignored for the same previous reason. These variable identifiers if they were not already resolved in the previous sections are rather rare and therefore do not have enough context regarding type usage. A placeholder type of null is used.

The priority is set to the lowest value of one as only one result of said abbreviation and (placeholder) type exists.

5.19 Combine and remove excessive entries

Due to the recursive resolving of abbreviations, there is a chance that entries exists that share the abbreviation, type, and full word. One possible bug arises whereby if two such similar entries exist, but with different priorities, the one entry with the lower priority will be loaded.

To avoid this, all entries that share these three attributes will be combined. The priority numbers of all similar entries will be summed up and the sum will be used in the single combined entry. Once that entry exist, entries that share the abbreviation, type, and full name will be deleted.

At the last step, any excessive entries will be removed. The plugin only loads up to three entries at any given time. The two attributes used to resolve the abbreviation are the abbreviation itself and optionally the type. Using this information, only the three most popular entries with the same abbreviation and type need to be stored and the remaining entries with the same abbreviation and type can be safely deleted because they will be never called at any given moment. This will also slightly reduce the file size of the exported abbreviation lookup tree data structure.

5.20 Serialization of the abbreviation lookup tree

After all these steps, the abbreviation lookup tree needs to be exported to a text file for later use. For every leaf node take the information from every parent node. The order is from the root node to the leaf node, left to right, and it is written as a string.

```
fn;str;filename;191
fn;Callable;function;258
```

Listing 5.16: Two example lines from text file, based of example of Figure 5.11

This is repeated until all leaf nodes are processed, this is then saved as a text file. The information is separated by commas. To recover the tree, the process is done in reverse, every line from that file will be read and a node is created from every semicolon separated value from a single line. One example can be seen in Listing 5.16, using the example from Listing 5.10.

In a single line from the exported file, the first item is the abbreviation, the second item is the type, the third item is the full result and the fourth item is the priority. All of these items are separated by semicolons and the lines are separated by a single newline character. This allows the exported file to remain barely human-readable. Its file size could have been shrunken with compression techniques but then the file becomes unreadable for humans and decompressing the file also takes up some time. Using standard settings the file takes about 3.3 MB of hard disk drive space.

5.21 Resolving the abbreviation

This subsection, unlike all previous subsections is done on local code with the lookup tree structure. Once the abbreviation lookup tree data structure is complete, it should be possible to resolve any abbreviations in local code that have at least two alphabetic characters. The tree data structure acts like a large lookup table from that point onward. The Python code corpus is not required for this section and the next one.

An existing variable identifier with at least two alphabetic characters is mandatory. A type hint that is assigned to that variable name is optional but recommended. If necessary, its type can be inferred to create a type hint. If the identifier is a plural or has numeric suffixes then these suffixes have to be removed and later re-added as of section 5.11. The function name is also required if the Type4Py type inference [50] is used. If a different type inference method is used that does not require the function name then it could be ignored.

From the root of the tree, it checks if a matching abbreviation exists at all in the first level of the tree. If no matching abbreviation is found, then the search stops right there and returns nothing. It will attempt to treat it like a single-letter variable and try to replace it outright with its own separate algorithm described in section 5.23. If a matching abbreviation is found then the search continues from the node containing said matching abbreviation.

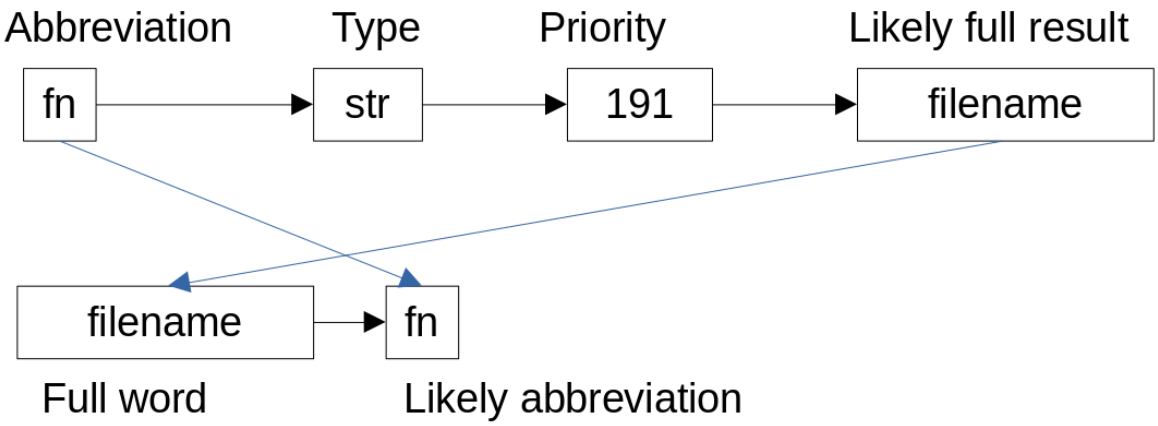
If a type is specified and a matching type is found, then the priority queue of the node containing that type will be looked up. Take the three results with the highest priorities. The user must then pick one out of these three results. The best one is marked.

If no type is specified or no matching type is found, than all subtrees from each type node are combined into a single tree. This means that the priorities of each full name under the abbreviation node are summed up for each full name node. The type will be ignored then. After that, the same procedure as described previously applies.

5.22 Shortening very long variable names

Using the abbreviation lookup tree data structure it is also possible to apply this in reverse, allowing very long variable names to be shortened. It is done on local code. PEP 8 [24] does not have a recommendation regarding maximum variable name length, but Pylint [71] does have a default maximum name length of 30. If this limit is exceeded then a warning will show up.

Typically these very long names consist of multiple English words, which are separated by underscores. If they are separated by capital letters then the same conversion process described in section 5.8 is used. A reversal of a single leaf of the abbreviation lookup tree data structure is seen in the figure below.



Fig, 5.17: Reversing one leaf of the abbreviation lookup tree data structure

In order to obtain an abbreviation from a single word, every entry in the abbreviation lookup tree data structure is searched that contains that word as the full result. The frequency is ignored because the frequency was how often the full result was used for the abbreviation, not the other way around, that frequency only works in one direction.

The type is also ignored because the existence of a matching type in a variable name containing multiple different words is not guaranteed. It is also irrelevant for this purpose and is not a hard requirement.

Initially the abbreviation is unknown. From there, its abbreviation is taken, and it is presumed to be the most meaningful abbreviation. If the next entry contains a longer abbreviation than the current presumed abbreviation, then that abbreviation replaces the current abbreviation as the presumed most meaningful abbreviation.

Another tiebreaker, if the length is identical is if the letters of the abbreviation are identical with the first letters of the full word. In that case the latter is taken. The abbreviation must never be identical to the full word.

This will be repeated until all entries are processed. If not abbreviation is found at all and the full word has at least five characters, then 50% of the first characters in the full word are used, rounded up, as abbreviation. The reason why longer abbreviations are preferred is because they retain more information from the full word than shorter abbreviations. Longer abbreviations are far less ambiguous than shorter ones.

Because said variable names contain multiple words, each individual word uses the same algorithm as just described above. It stops when the total string length is below 30, which is under the soft limit in Pylint, or when all words in the identifier are processed.

The number of very long variable names in Python is very low, so this is less of an issue in practice, unlike in other programming languages that use more verbose names.

5.23 Overview of single-letter variable replacement

This is an overview of the single-letter variable replacement that will be explained in the remaining sections of this chapter. The basic idea was already described by Tran et al. in their paper about how to recover minified variable names which are often reduced to an unrecognizable single letter or two [22].

The rationale here is to treat these single-letter variable names as if they are fully unknown and try to “recover” the full variable name.

The preprocessing step is identical and is already described in section 5.7. The same applies to finding variables which is already described in section 5.8 in case for variables on the left side of the assignment or in case of variables on the right side of the assignment the steps are already described in section 5.15.

New to this is the creation of variable relation lists that describe all actions that are done with a given variable. The description on how to create these actions which are then added to a variable relation list are written in section 5.26.

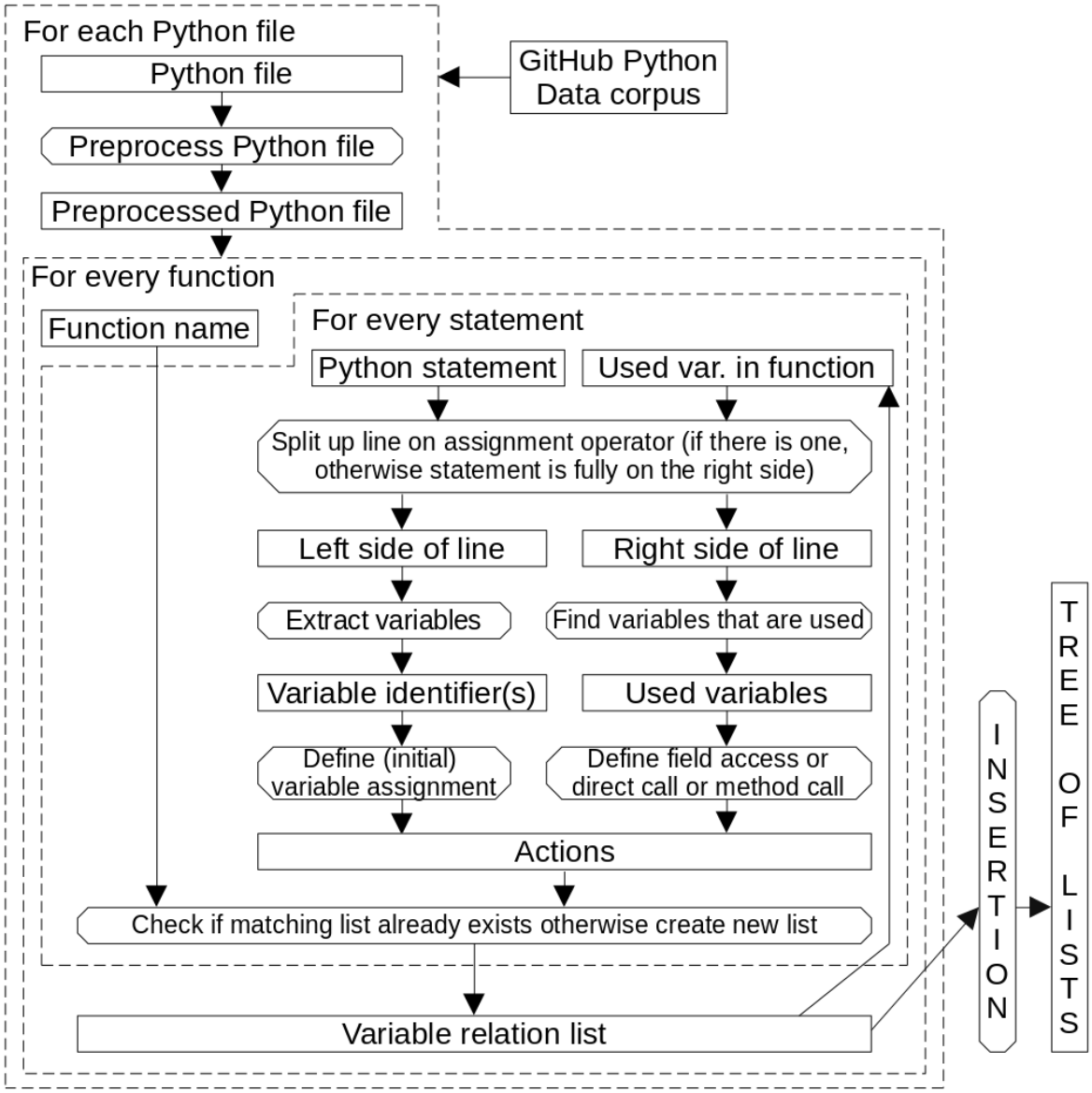


Figure 5.18: Simplified single pass of single-letter variable replacement

The details on the variable relation list data structure itself is written in the next section. These will be, once the end of the file is reached, be added to a tree data structure for faster access. That is described in section 5.25.

Not seen on Figure 5.18 is that the tree needs to be exported to a file structure. First it needs to be serialized, then it needs to be processed to a tree which then is exported This is described in section 5.27 and 5.28 respectively.

The actual comparison of the local variable relation lists in the local file and the other lists in the tree data structure is described in section 5.29. All removed features, including some other features unrelated to the single-letter variable replacement is described in section 5.30. The heuristic is the similarity of the variable usage context.

5.24 Single-letter variable relation list

The entire single-letter variable replacement algorithm is based of an existing algorithm named JSNEAT by Tran et al. [22]. However, that was originally meant for JavaScript code and therefore must be adapted to Python code. Unlike in the previous sections, type inference is not used. The data structure consists of two attributes and one single variable relation list.

The first attribute is the variable name itself, which is the name that would replace any single-letter variable name.

The second attribute is the function name from which the variable originated in the initial list creation. If the variable name is a global value, then it is named as such.

At the core level there is a linked list data structure. There are four categories which describe what action is done with the variable in that function by a certain object. If one such interaction occurs, this will be written to the list, in the order they occur, from the first variable assignment until the function ends.

Assignments are direct assignments to the variable. This is always the first interaction with the variable, because every variable needs an initial variable assignment. This will always be used every time a variable has been reassigned. For function parameters, as they do not have the initial variable assignment, one action is described for the initialization of the parameter in the function. The objects are the variables on the right side of the equations.

Direct calls are actions when another different variable in the same function calls the variable directly, not one of its methods or fields. For an example, it is a direct call if the variable is used for another variable assignment for a different variable.

Method calls are actions similar to direct calls except that one of the methods from that variable are called instead. This can also occur if the method is called by itself in the function without any variable assignment to a different variable

Field accesses are actions that refer to one of the variable fields.
The objects for the latter three actions are the variables on the left side of the equation if there are any, otherwise it is declared as null because there is nothing.

One more optional set of data can be used, but this has been left out due to system memory constraints. It would store any additional variable relation lists of all variables in the same function. This is to obtain the multi variable context from the same paper [22]. However, due to said constraints this is optional and is reduced to just the variable names and types of the other variables in the same function. The type was also considered to be used as another attribute but this was not deemed to be necessary.

An example for three variable relation lists for each variable can be seen in Table 5.21, which refers to Listing 5.20.

5.25 Single-letter variable replacement tree structure

Due to the very large number of variation relation lists in order to find the correct lists quickly, there is a two-layer tree before the list can be accessed.

The first level denotes the size of the list by its number of elements in a list. This is because a better result is more likely the closer the size of the lists are. If the lists were identical then they also have to be identical in size.

The second level is the function name from which the lists are associated with. The rationale is that certain function names or subtokens such as “get” or “set” are more likely to yield more accurate results. An example can be seen below.

Past these level one or more lists as described in the previous section can be found. Each list then contains a name attribute, which indicates the likely variable name for a given table. The lists are the leaf nodes and no further levels exist. The name attribute is inside the list. Further details are in the next section.

Internally, this is realized as a nested hash map if it is loaded to system memory or as two levels of directories and text files if stored on the hard disk.

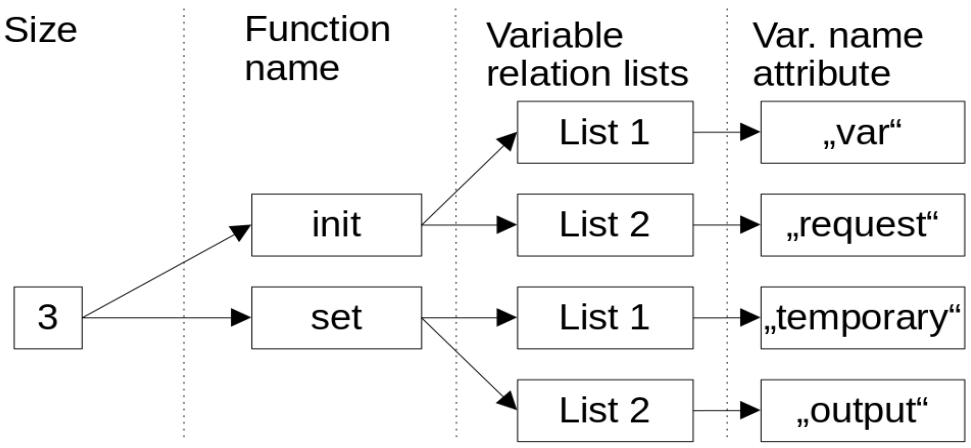


Figure 5.19: Single-letter variable replacement tree structure

5.26 Building the variable relation lists

The entire Python code dataset which is already described in section 5.3 will be used again and every Python file in it will be processed to build these ordered lists that are described in section 5.24. This section is based of the previous work by Tran et al. [22].

One important aspect is to respect the scope of the variable. A variable that is defined inside a given function is only valid inside that function. Therefore, a variable with the same name but in a different function is not the same previous variable in the previous function. It is possible to give a local variable the same name as a global variable, but this is discouraged.

Boundaries for a given function must be defined. In other programming languages curly braces (“{“ and “}”) exist to define the start and end of a function, but Python does not use these. Instead it uses non-empty whitespaces to define a function. The number of whitespaces before a function declaration are counted in Python. The function ends if new code is found that have the same number of whitespaces or less than the number of whitespaces for the function declaration [81].

```
def addition(
    param_1,
    param_2
):  # not the end of the function

    variable = param_1.x + param_2.get_y()
    return variable

# comments do not count
# line below marks the end of the previous function
def different_function():
    return addition(param_1, param_2)
```

Listing 5.20: Python code showing where a function ends

Note that a multi-line function declarations are internally treated as a single line so the end of such a statement does not mark the end the function itself. Comments also do not count since comments are completely ignored during runtime.

From there on, inside the given function, as soon as a parameter or variable is declared for the first time, a new list will be created for that parameter or variable. The name of it will be used as an attribute for the variable name. The first action, which is always a variable assignment, will also be written down. In case of parameters, they will be noted as parameters. From there on, if one of these variables occur again, either as another variable assignment by some other variable or is being called by a different variable, these actions will be noted as well. In case of being called by a different variable, it is being differentiated as either a direct call when the variable itself is called, a function call when one of the functions of the variable is called or a field access when one of the inner fields of a variable is called.

The object is a variable name from the right hand side of the variable assignment in case of such an assignment or a variable on the left hand side in the other actions. More details are in section 5.24. An example is given in Table 5.21 for the Listing 5.20.

Variable name	param_1	Variable name	param_2	Variable name	variable
Function name	addition	Function name	addition	Function name	addition
Action by	Object	Action by	Object	Action by	Object
Assignment	[param]	Assignment	[param]	Assignment	param_1.x
Field access	variable	Method call	variable	Assignment	param_2.get_y()
				Direct call	[return]

Table 5.21: Variable relation lists for Listing 5.20 in the function “addition”

Once the function ends, the function name will be noted, the lists for that function will be saved and will not be altered further. This process then repeats for all functions in a single file until the entire file is fully processed. This is repeated for every file in the Python code corpus until all files are processed.

In parallel, each list is assigned to a position in the tree structure. The parameters for the tree structure is the size of the list and the function name. The list will be placed accordingly to the size node, then the function name node. For details see section 5.25.

5.27 Serialization of the variable relation lists

The lists require to be serialized in order to be written to the hard disk drive.

A single list starts with a single “[VAR]: “. At the same line, the variable name that is associated with the list is written. No semicolon is used here unlike in the other lines, this is to differentiate it. If another line is encountered that uses the same “[VAR] :” string, then the previous list ends and a new list starts.

Inside the list, the actual action is written on the left side before the semicolon and the object is written on the right side after the semicolon. Each entry is separated by a newline character. The actions themselves are always fully capitalized with the words being separated by the underscore character.

Each saved file can only store up to 100000 lists in order to prevent the files from becoming to large. When a file is about to become to large, a new file will be generated instead. This will not store any tree data structures but it can be reconstructed.

During the loading process, every file is loaded to reconstruct the lists. The serialization process is then simply done in reverse.

This will also restore the single-letter variable replacer tree structure by inserting every entry into a new tree.

It is also possible to only serialize a select number of lists with a certain criteria. The criteria in this case are the most frequently used function names. The idea here is that the lists that use most frequently used function names should be stored in memory instead rather than on the hard disk drive, because lists that use one of the most frequent function names, such as “init”, “get” or “set” are called more frequently than the ones with obscure function names. There are also far more lists with such function names because said names are used more often. An example based of Table 5.21 can be seen in Listing 5.22.

```
[VAR]: variable
ASSIGNMENT;param_1.x
ASSIGNMENT;param_2.get_y()
DIRECT_CALL;[return]
FUNCTION_NAME;addition
```

Listing 5.22: Serialized output of a single list from Table 5.21

5.28 Exporting the tree data structure

Storing all entries in system memory can consume a very large amount of system memory. In order to not use too much system memory the tree data structure can be saved to the hard disk drive instead.

For every node on the first level of the tree structure which is the number of entries in the table, a directory will be created for every node. From that directory which represents the first level, all child nodes of the node will be represented by their own subdirectory. The first and second levels are not leaf nodes, but the third level acts as the last level. The tables are stored as text files and all attributes, including the name attribute are found inside each table.

In practice, the system will attempt to access the cached list entries in system memory first. If nothing is found in system memory, then the exported tree data structure on the hard disk drive will be accessed instead, which can take a few seconds longer. Therefore less frequently used data is stored on the hard disk drive.

The main drawback is that due to the large amount of small files, it can take up a surprisingly large amount of hard disk space. Using the Python code dataset, it takes up 22.2 GB of hard disk space, compared to 7.5 GB of hard disk space if all tables are stored as a single file or as a small number of large files. This may vary depending on the file system block size, which dictates a minimum file size per file.

5.29 Finding a replacement for a single-letter variable

To find a replacement name for a single-letter variable, the local code needs to be analyzed first and its own lists have to be created, in a similar fashion that is already done in the Python code dataset as seen in section 5.26. It is similar to the work by Tran et al. [22].

The local lists will ignore its own variable name attribute, because that will be replaced. Said local lists will then be compared to the large amount of the lists that were generated from the Python code dataset. To speed the process up, only lists will the same number of entries in the list will be compared and if available, only those that share the same function name or at very least contains part of the function name. This information is then used to access the lists via the tree data structure.

From there on, one entry from the local list will be compared to one entry from the other list. A tally is used to track the number of list mismatches. For every entry mismatch, the tally is increased by one. Therefore, a lower number is better. This is then repeated for every entry in both tables, in order. This means that the first entry in the local list is compared to the first entry in the other list, the second entry is compared to the other second entry, and so on. This is also done to reduce the amount of time taken.

Once this is finished, the tally is placed with the associated variable name of the other table from the Python code dataset in a minimum priority queue, using the tally as the key. If a better result is found with a lower tally, then it displaces the older results.

Typically it is done in the correct order in the list, but the order can also be ignored. Once all the relevant lists are processed, the priority queue is then polled up to three times to get the most likely variable name replacements for the single-letter variable. Another filter can be implemented to filter out any results that result in another single-letter variable. The user then picks one out of three results and then the single-letter variable name is replaced by the new name.

5.30 Removed aspects

5.30.1 Removed aspects in abbreviation expansion

At some point, regarding the abbreviation expansion, any words from the related comments and documentation of the code, in this case the comment next to the actual code and the documentation of the function, were also considered to be used as potential full words, as this was already done in the original paper by Lawrie et al. [21]. In practice however, this caused inaccuracies, as commonly spoken English words commonly used in typical English speech which were found in documentation, were picked up as potential full words to resolve the abbreviation. This effect was magnified by using a large Python code dataset.

At the same time, because a large code dataset was used, it simply was not necessary anymore to take comments and documentation into account.

As already mentioned in section 5.17, a part-of-speech tagger was not implemented because it does not work on single words.

A spell checker was supposed to be integrated in this algorithm, in the event of any misspelled word. However, that would have to be done at the integrated development environment (IDE), and its own spell checker is too tightly integrated. The other reason is that an IDE usually has a spell checker. In that case when the IDE marks a misspelled word, this needs to be corrected first before using the algorithm. Even then,

5.30.2 Removed aspects in single-letter variable replacer

In the single-letter variable replacement type inference was considered and tested. Not only is the time performance unacceptable, but the accuracy significantly decreased if the inferred type is taken into account.

The same applies for any variable names that are in the same function as the specific variable name. Checking for any related variable names in the same function as that variable name, then comparing both lists. Even if this was only used as the last tiebreaker between otherwise similar lists, the accuracy also decreased significantly.

One possibility on why this happens is that at that point the comparison becomes too specific, and no match can be found. Apparently the lists must not be too specific in order to have some amount of accuracy. Basically, overfitting should be avoided.

Chapter 6

Implementation

In the sixth chapter, the architecture and the technical implementation details are explained.

There are three separate programs. The first program is the integrated development environment (IDE) plugin, the base platform is explained first followed by the architecture of the plugin.

The second and third program are command-line programs solely for the creation of the resources necessary for the plugin which is explained in section 6.3.

The four resources themselves are described in detail after the programs are described. At the end of the chapter the setup and graphical user interface (GUI) of the plugin is shown.

All programs are written in Java, but the programs analyze Python code. These cannot analyze other programming languages, including Java. The reason is that most Data science projects are written in Python, as already mentioned in Chapter 1.

6.1 IntelliJ Platform Plugin

The IDE for which the plugin is developed for is PyCharm by JetBrains, which is based off the IntelliJ Platform by JetBrains [82]. The main difference is that PyCharm is for developing Python, while IntelliJ is for developing Java. This IDE already supports many features, such as syntax highlighting, spell checking, simple variable renaming, debugging, basic code generation, version history and so on. The IDE and its plugins are written in Java.

Its functionality can be expanded with plugins. Plugins can add new features such as different user interface (UI) themes, better integration with commonly used frameworks, better support with different programming languages, integration with build tools and revision control and so on. This IDE and its plugins are written in Java and runs on the Java Virtual Machine. Therefore, this plugin is also written in Java. The creation of a new plugin can be simplified by using an existing template that is found in GitHub by JetBrains themselves. [83]. The plugins are then distributed on the JetBrains Marketplace where users can download and use these plugins.

To test the plugin, a new instance of the IDE is launched but with the plugin installed while the current IDE that is used for development is logging the different instance of the IDE with the plugin.

6.2 Variable name replacer plugin architecture

This section describes the end-user client architecture used in the actual Variable Name Replacer plugin which is summarized in the figure below. As the name suggests, this requires the PyCharm IDE by JetBrains [82]. Being a plugin of that IDE, it is written in Java. The basic architecture of the plugin is depicted below in Figure 6.1.

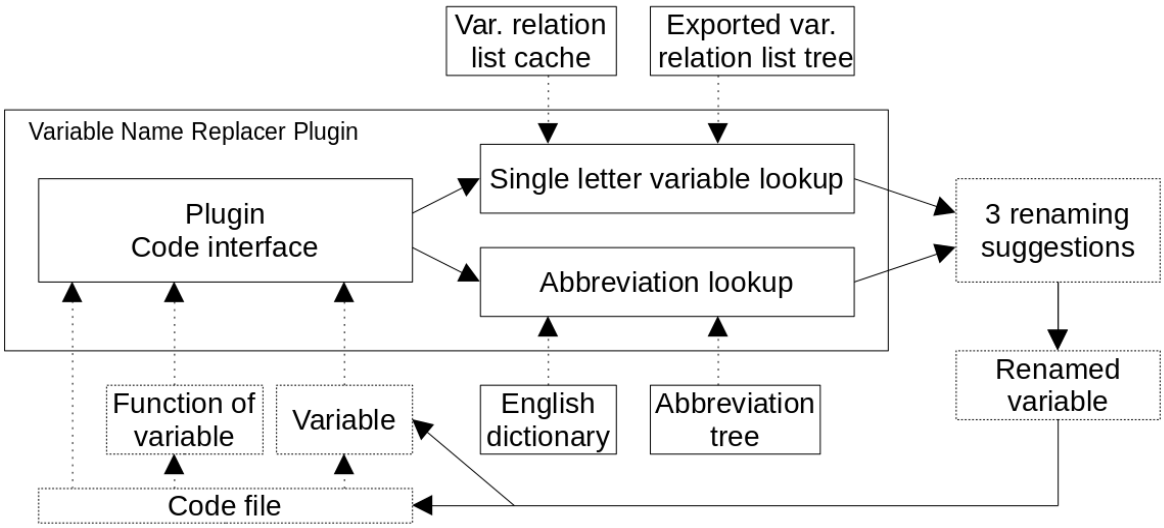


Figure 6.1: Variable name replacer plugin architecture

The variable relation list cache, the english dictionary and the abbreviation tree are the resources that are loaded into system memory when PyCharm is launched with this plugin. The exported variable relation lists is the only resource which remain loaded from the hard disk drive. All of them and how these were generated were explained in Chapter 5 and are required for the operation of the Variable Name Replacer plugin.

There are three parts of the plugin. One limitation of this plugin, due to limitations of the IDE is that it only works in files that belong to the project that is opened in the IDE. External files do not support refactoring in the IDE, which also prevents the plugin from working, since it relies on internal refactoring functionality from the IDE. Input data in all cases is the variable name that will be replaced, its associated function name and the code file it is located in.

6.2.1 Plugin Code Interface

The plugin code interface is the component that acts as the outer layer of this plugin. It interface with PyCharm directly in order to read the Python file. This component reacts when the user right clicks on a token on the code, which in PyCharm is internally referred as “PsiElement”. Each “PsiElement” contains an internal navigation element. Variable names are called “PyTargetExpression” in PyCharm, internally, while parameter names are named “PyNamedParameter”. These are the navigation elements to look for.

If the token is not a variable or parameter name, which internally means that it is not a navigation element named “PyTargetExpression” or “PyNamedParameter”, then the option to replace the variable name will not show in the context menu that shows after the user right clicks on an element, because it can only replace variable and parameter names.

When the user right clicks on a variable or parameter name, then the option to replace the names will show up. Once that option is clicked, the variable name is taken. The plugin then checks if the word has at least two alphabetic characters. If it meets that criteria, the name is then passed to the abbreviation lookup.

Otherwise it is a (pseudo-)single-letter variable and gets passed to the single-letter variable lookup instead. Either way, up to three renaming suggestions will be shown to the user. Once the user picks one, all variable names within a given limited scope of the original name, usually inside a function, will be replaced with the picked suggestion in the code.

6.2.2 Abbreviation lookup

The name is passed through the abbreviation lookup. It first checks if the name is already a full English word by checking the English dictionary. If this is the case, then no further processing is done and there will not be any renaming suggestions since it is not possible to expand an English word further by the existing algorithm.

Otherwise the name will be looked up in the abbreviation tree to check if there is an exact matching abbreviation for the exact name and from there on find up to three matching full English words for the given abbreviation. If there is a type hint, that type hint will be taken into account, otherwise the type is ignored.

Type inference is currently optional and depends whenever there is a type hint at the variable name. PyCharm has its own type inference, which can be used to generate the type hint, that can then be used in the lookup. Using Type4Py [50] is too resource intensive for some computers and the response time, which would be nearly instantaneous, would take seconds to process by the Type4Py plugin. The exact details on how an abbreviation is resolved, are described in section 5.21.

6.2.3 Single-letter variable lookup

In that case when the variable name does not have at least two alphabetic characters, the single-letter variable lookup is performed instead.

First, local variable action lists are created, then the plugin tries to get the closest match between the local variable action list of the variable name that is to be replaced and the list cache that is stored in system memory. The cache only contains lists that use some of the most frequently used function names such as “init”, “get”, or “set”. If the function name of the variable is very frequently used then the cache is called first.

If no lists with a matching function name is found, then the exported list tree on the hard disk drive is checked instead. This is done to speed up the processing time because there are many more lists in the case of when they use frequently used function names, but fewer lists with obscure function names. The entire process is described in detail between section 5.23 and section 5.29. The end result are up to three renaming suggestions, as already described previously. Type inference is not supported for this operation.

Two sets of system requirements exist, depending if the user wants to create the assets on their own or if preexisting ones are used. If only the client plugin for the PyCharm IDE is used, then the requirements are relatively moderate.

6.3 Programs to generate resources

There are two separate programs that exist to generate the resources that are necessary for the plugin. These programs do not interact directly with the plugin directly and can be run independently. All programs do not have a graphical user interface (GUI), instead they run on a command-line interface. They are also written in Java.

Both programs require a large amount of existing Python code, such as the GitHub Python code dataset and an English dictionary file as input files. The output depends on what program is run. It is highly recommended to use pre-generated resources instead of manually generating these resources, as this is a time consuming procedure.

6.3.1 Abbreviation tree data structure generator

The abbreviation tree data structure generator generates the abbreviation tree data structure from the GitHub Python code dataset. The English dictionary is used to check for full English words. This is its only mode of operation. Once the generation is finished, a single text file that is the serialized abbreviation tree data structure is written to the hard drive. This file still needs to be loaded and deserialized into the plugin.

6.3.2 Single-letter variable tree data structure generator

Unlike the previous application, this one has three modes of operation. For this example, it is done with the GitHub Python code dataset.

The first mode of operation is to generate the variable relation list cache which only stores the most frequently used variable relation lists. After the generation these lists are directly stored as a small number of text files in the hard drive. For this example it takes about 2,4 GB of actual hard disk drive space.

They are expected to be loaded directly into system memory. Even when the size is smaller than the all variable relation lists, it still takes up a lot of said memory. The plugin which has to load the cache into needs to allocate 8 GB worth of system memory, which is still a lot when many computers still have about a high single-digit to low double-digit amounts of RAM. The computer needs 16 GB of RAM to run the plugin at all.

The second mode is the first part of the two-step process. It operates similar to the first mode, except instead of creating a small subset of variable relation lists, it creates all variable relation lists and exports them to a small number of larger text files to the hard drives. It takes about 7,5 GB of actual hard disk drive space.

It generates so many lists that storing them on system memory is not feasible for the plugin. If all lists were loaded in system memory for the plugin, then the computer would at very least require 32 GB of RAM, with very little headroom after that.

The third mode takes these files generated from the second mode and exports them into a tree-like file directory structure, similarly modeled after the single-letter variable replacement tree structure described in section 5.25, where each directory level represents a node in the tree. This will not be loaded into system memory, instead these infrequently used variable relation lists are accessed directly from the hard drive. The main drawback is that it takes up much more real hard disk drive space compared to the other modes. It requires 22 GB of hard drive space and depending on the size of the minimum cluster of the file system, it can be more, because are 5907661 tiny files and 6145823 folders.

Some Java code for the single-letter variable tree data structure generator, specifically from the second mode, is reused in the plugin to generate the local relation lists in the local code files, then said lists are matched with one of the lists in the global single-letter variable tree data structure to replace the single-letter variable.

6.4 External resources

For the operation of the plugin multiple resources are required. The first one cannot be created by any algorithms. Three more prerequisites are required for the initial creation of the other three resources. One listing is provided as an example line for every subsection.

6.4.1 English dictionary file

The first resource is a text file that contains full English words. These English words are only separated by newlines (“\n”). This file can be commented using a single hash character followed by an exclamation mark. It is therefore easy to add or remove words if needed. This file cannot be algorithmically created, it is therefore obtained elsewhere, using the Wiktionary top 100000 most frequently-used English words as the baseline [65]. Some obvious English words were still not detected initially, so they were added manually to the list while obvious non-English words, such as French words commonly used in the English language, are removed. More details in section 5.2.

```
#!comment: This is a comment
word
```

Listing 6.2: Two example lines from the text file that contains English words

6.4.2 Abbreviation tree file

The fourth resource is the exported text file from the creation of the abbreviation tree. Which is generated from the abbreviation tree data structure generator from subsection 6.3.1. Every line contains a leaf node, whose full details are explained in section 5.6.

```
abbr;str;abbreviation;1000
```

Listing 6.3: One example line from the exported abbreviation tree

6.4.3 Variable relation list cache

The second resource is the subset of the most frequently used lists of all the lists required for the single-letter variable replacement algorithm. That subset acts as a cache and is loaded directly into system memory for faster access. If it was loaded from the hard drive, loading large frequently used variable relation list would take an unacceptably long time, about a minute, for every operation with a variable relation list. This is generated from the single-letter variable tree data structure generator, see the subsection 6.3.2 for details.

```
[VAR]: variable
ASSIGNMENT;param1.x
ASSIGNMENT;param2.get_y()
DIRECT_CALL;[return]
FUNCTION_NAME;addition
```

Listing 6.4: Example lines using example from Table 5.21

6.4.4 External variable relation list tree data structure

The third resource are the exported list tree required for the single-letter variable replacement algorithm. These are stored on the hard drive and are not loaded into memory during runtime. These exported variable relation lists are used infrequently and are very small in size so loading a small variable relation list only takes less than a second. This is generated from the single-letter variable tree data structure generator, see the subsection 6.3.2 for details.

6.5 System requirements

It needs to run the IDE itself and the plugin. A solid state drive is recommended in general to improve the overall user experience on the computer regardless of the program.

	Minimum	Recommended
CPU	x86-64 2.0 GHz dual-core processor	x86-64 2.0 GHz quad-core processor
RAM	8 GB RAM	16 GB RAM
Storage	500 GB hard disk drive	500 GB solid state drive
Internet	Broadband internet connection	Broadband internet connection
OS	Windows 8+ or Linux w/Gnome	Windows 10+ or Linux w/Gnome

Table 6.5: System requirements for running the PyCharm plugin

The minimum requirements as seen on Table 6.5 are the absolute minimum to run these programs at all or else these programs will not run at all. The user experience is not guaranteed to be usable for day-to-day tasks in terms of usability if the minimum requirements are barely met. Therefore, the recommended systems requirements exist for optimal user experience. A 64-bit processor of the x86-64/AMD64 architecture and a 64-bit operating system is required as more than 4 GB is used which is the upper limit of memory that 32-bit CPU architectures can address [84].

At the bare minimum a dual-core processor is required. For optimal use a quad-core processor is recommended.

Currently ARM-based processors are not supported. Theoretically if the libraries are already ported to ARM it could be possible in the future to support the latest ARM-based computers, but currently it is out of question. This also means that the latest macOS devices that are based on ARM are not supported. The x86-based macOS has not been tested due to lack of available hardware. The requirements are much higher if the abbreviation tree and the relation lists are created manually as seen on Table 6.6.

Due to the presence of very rare but non-standard characters used in some filenames in Git repositories, it cannot run on Windows. More storage is required to store the Python code dataset. 32 GB of RAM are required to run the processes to create these and more are recommended for a more smoother process.

A faster processor is also recommended to speed the entire process up, but a higher core count will require more system memory as using more threads require proportionally more system memory. This also includes running any external programs required for the process of creating these prerequisites.

	Minimum	Recommended
CPU	x86-64 2.0 GHz quad-core processor	x86-64 4.0 GHz octa-core processor
RAM	32 GB RAM	64 GB RAM
Storage	2 TB hard disk drive	2 TB solid state drive
Internet	Broadband internet connection	Broadband internet connection
OS	Linux	Linux

Table 6.6: System requirements for creating resources

The cache would not be required at all if all variable relation lists can be loaded into system memory but it is not practical yet as stated above.

6.6 Setup for plugin

To install the plugin, it has to be installed manually from the hard drive. Currently, it is not on the JetBrains Marketplace [85], it is found on the supplementary disc.
First, the user must click “File” on the menu bar, then select the “Settings” item.

The settings will show up. From there, the user has to clock in the “Plugins” item on the left side of the settings. Then the user has to click on the cog icon next to “Marketplace” and “Installed”. A context menu will show up. The user then has to click on “Install Plugin from Disk”. A file chooser will appear where the packaged zip file of the plugin named “editor-0.0.1.zip”, which is located on the local hard drive, must be selected. The user is then prompted to restart PyCharm. Figure 6.7 shows the steps of the installation.

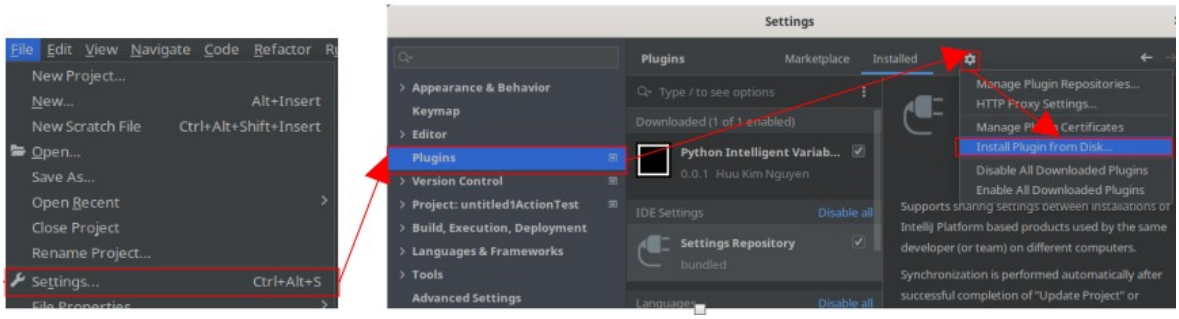


Figure 6.7: Instructions on installing the plugin in PyCharm

The setup is not done yet. In the settings, a new item will appear in the category “Tools” named “Variable Replacer Settings”. Clicking on it will show four empty text fields that have to be filled out before the plugin becomes functional. Clicking each corresponding button shows a file chooser. The user picks the correct folder or file to fill out each field. Figure 6.8 shows a screenshot of these settings.

Once all text fields are filled, the memory size must be increased from 2048 MB to 8192 MB in the VM settings in PyCharm. To achieve this, click on “Help” in the menu bar and from there select the “Change Memory Settings” item. A popup will show up, the value has to be changed to 8192 MB. Then confirm the change which also restarts the IDE.

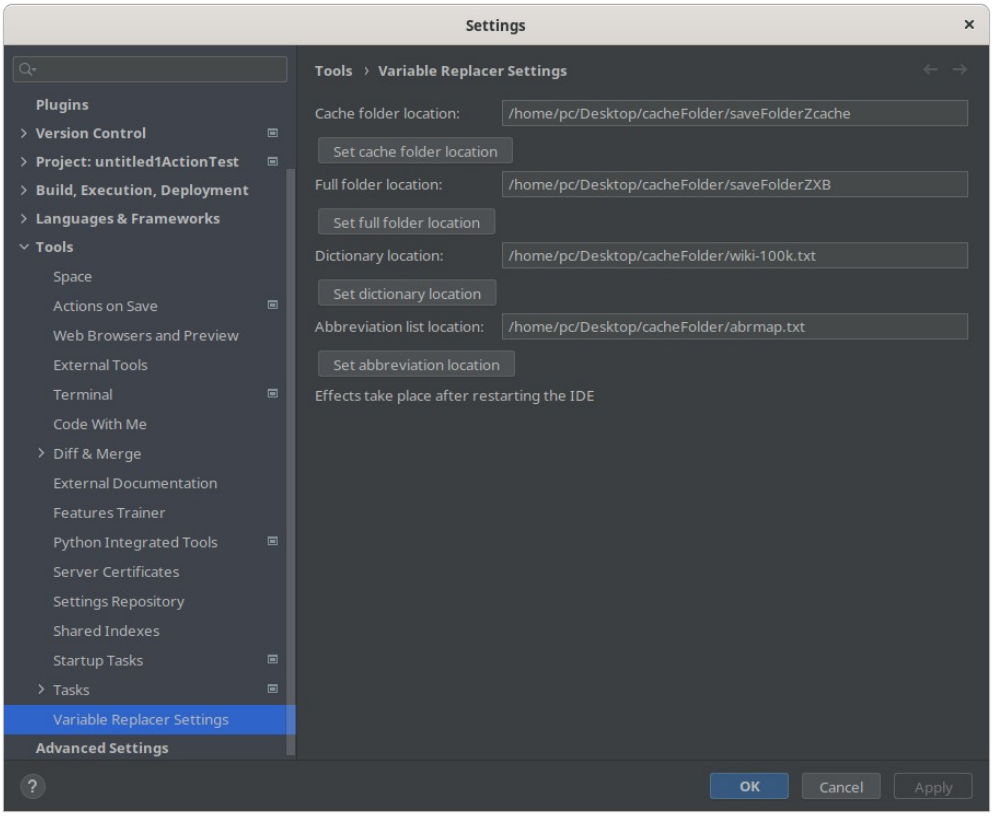


Figure 6.8: Variable Replacer Settings

6.7 Plugin usage

After PyCharm is launched, the cache is loaded in the background, which depending on the computer, may take up to one minute to load. While it is loading, the IDE can be used as normal, but the renaming functionality cannot be used until it finished loading.

To intelligently rename any variable, the user has to right click on a variable name or on a parameter name, then click on “Rename Variable Name” in the context menu. In case the plugin is still loading, said option is grayed out and replaced with “Cache is still loading or Settings are not set”. Which means that this also occurs if the setup as mentioned in the previous section is still not complete. In Figure 6.9, which is an example, the user right clicks on the name “var”, which then shows that popup window.

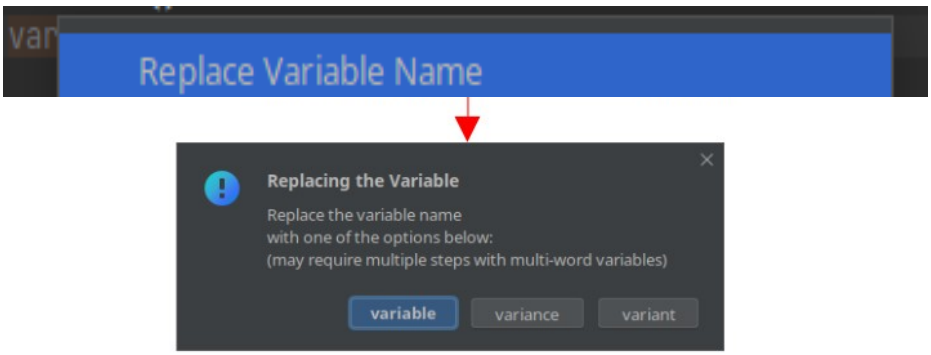


Figure 6.9: Resolving a variable named “var”

Any other variable names or words in the code are not compatible with the plugin. If the user tries to right click on them, the option to rename these variable names will not appear at all.

Once this menu item is clicked, the plugin processes the variable name. It only requires up to three seconds at most. When it is finished, the user can choose from one of the up to three renaming suggestions. The best renaming suggestion is already highlighted and can be chosen by just pressing the Enter key. Alternatively the user can just close the window to cancel this action.

All occurrences of that variable will be renamed when the user clicks on one of the buttons. If an inner variables shadows an outer variable, and if the user wants to rename the inner variable, then only the occurrences of the inner variable will be renamed and vice versa. It will not rename any variables that are commented out.

If the variable name contains multiple abbreviations that cannot be resolved together, but can be resolved individually, the plugin only extends one abbreviation at a time, from left to right. The user therefore has to “rename” the variable name multiple times before all abbreviations in the variable name are expanded.

One fundamental limitation due to PyCharm is that it relies on the internal search function to find these variables. Therefore it will not work on externally loaded files. To get around this, the files can be moved to the project directory so it is recognized as an internal file.

A type hint can only be taken in consideration if it exists. If no type hint exists the replacer will not take types into account. Type hints can be added by PyCharm by right-clicking a variable, then select the “Show Context Actions” item, then select the
“Add type hint for variable” item to add the type hint.

This must not to be confused with the existing ability of simply renaming an existing variable or the ability to let the IDE rename all instances of said variable with the existing refactoring tools that are built-in the IDE.
This process should be near instantaneous, unlike the setup which takes a long time.

Chapter 7

Evaluation

In this chapter multiple evaluations are being done. First the testing environment and its hardware choices are being detailed. For the majority of the evaluation an external linter named Pylint [71] was used to evaluate the mean code quality on multiple Python code datasets, which includes one specific to data science projects and another one without said data science projects.

Also investigated is what variable names the most popular and how accurate the abbreviation expansion algorithm is and the single-letter variable replacement.

This is then concluded with the evaluation of how many variables can be expanded to more descriptive names and how long each process takes in terms of time performance. All the research questions will be answered at some point during the evaluation.

7.1 Testing hardware

For all evaluations this hardware was used on Table 7.1.

OS	Debian 11.1 “bullseye” (Linux 5.10.70-1, AMD64)
CPU	AMD Ryzen 7 3800XT (8 cores, 16 threads, 4.4 Ghz)
RAM	32 GB DDR4-3600
Graphics card	NVIDIA GeForce GTX 960 4 GB
Primary SSD	Western Digital WD_BLACK SN750 NVMe SSD, 1 TB (PCIe 3.0)
Secondary SSD	Crucial MX500 SATA, 1TB (SATA)
Hard disk drive	Western Digital WD10EZEX-75M2NA0, 1 TB (SATA)
File system	ext4, default settings
Internet bandwidth	50 Mb/s downstream, 10 Mb/s upstream

Table 7.1: Specifications of the computer

All evaluations are done on the primary solid state drive (SSD) unless specifically noted, except for one evaluation which is specifically done to test the time performance difference between a solid state drive and a hard disk drive (HDD).

The graphics card is solely for video output only because the CPU does not have any integrated graphics unit. For all the evaluations, the graphics card was not used for any computational purposes.

All of the evaluations will currently not run on Windows because the file names in a very small number of projects in the datasets are forbidden in Windows because they use characters or reserved names that NTFS does not allow, which meant that a Linux-based operating system was chosen.

The reason to choose ext4 as the file system is that this is the default option that most users use. Other file systems as btrfs [86], XFS [87], and ZFS [88] exist, but btrfs is still not considered to be finished and is therefore not 100% stable. The actual differences in terms of I/O performance are too small to be significant.

No macOS computer was available for testing. Even though macOS is a UNIX-based operating system, one particular complication could arise because macOS is currently in the middle of a transition from the x86 to the ARM architecture. All evaluation tools were written in Java 11, but all projects in the datasets use Python 3.

7.2 Pylint

Pylint, originally by Thénault and now maintained by the Python Code Quality Authority [71], is a static analysis tool for the Python programming language. This software can be used to detect errors and warnings in the code and check if the PEP 8 naming conventions are being followed. It settings can be changed by altering the .pylintrc file which can be located at the root of the project directory or in the users home directory.

There is a very large number of possible warnings in Pylint, but most of them occur too rarely to be significant. Not all warnings correlate to code quality. Because of this, for this evaluation, only 15 warning types out of 354 types are taken which are the most frequent and correlate to code quality.

One counter example, which appears in both data science and non data science datasets are missing imports. This is likely caused by missing dependencies on the local computer and it is assumed that the libraries that the programmers use are loaded externally and are not in the repositories. Therefore this is left out. It also does not affect the actual code quality.

The default .pylintrc file has these default settings. If one of these are not met, a corresponding warning will be shown. One important aspect is that the identifier name rules, such as the minimum length of 3 and the maximum length of 30 cannot be altered.

The global Pylint configuration can be overridden by a local configuration file, either as another .pylintrc file or inside a setup.cfg file. For this evaluation, all local configuration files are deleted so that the global configuration is always used.

Multithreading can be optionally enabled. However, Pylint has an extremely low chance of not being able to fully process a single repository, which can already take hours in some instances. Instead, multiple Pylint instances are run simultaneously. The repository will be excluded completely if it cannot be processed by Pylint. Related to that, there is also a chance of a memory leak, most of the time it should not use more than 1 GB of system memory, but some repositories use more than 20 GB, which is indicative of a leak.

There are five prefixes for warnings, not counting I(nfo), from the manual pages [89]:

- C(onvention), which are warnings that violate the PEP 8 code conventions [24]
- E(rror), which indicate actual runtime errors if this was run in the interpreter.
- R(efactor), which are general spots where refactoring should be done.
- W(arning), which indicate actual Python warnings.
- F(atal), which are fatal errors that also stops Pylint from processing any further.

Maximum of, according to manual pages [89]:

- 5 arguments in function (otherwise triggers R0913)
- 7 attributes in class (otherwise triggers R902)
- 5 boolean expressions in an if statement
- 12 branches (if/elif/else) in function/method
- 15 local variables in in function/method
- 7 parent classes for a subclass
- 20 public methods for a class
- 6 return calls in function/method
- 50 statements in function/method
- Line length of 100 (in contrast to the PEP 8 [24] recommendation of 79)
- 1000 lines total per module (file)
- No single line class statements
- No single line if statements
- 4 spaces as indentation

Error/Warning code		Description
Non compliant name	**(C0103)**	Identifier is too short or long or uses wrong naming
Missing module docstring	(C0114)	No docstring in module
Missing class docstring	(C0115)	No docstring in class
Missing function docstring	(C0116)	No docstring in function
Line too long	(C0301)	Ling is longer than 100 characters
Trailing whitespace	(C0303)	Whitespace past the last character in a line
Wrong import order	(C0411)	Import order is not correct
Method could be function	(R0201)	A class method could be a “static” function instead
Too many instance attribut.	(R0902)	Eight or more class attributes (>= 8 self.x)
Too many branches	(R0912)	Cyclomatic complexity in function too high
Too many arguments	(R0913)	Six or more function arguments
Too many local variables	(R0914)	16 or more local variables in function.
Bad indentation	(W0311)	Does not use 4 spaces as indentation
Unused variable	(W0612)	Variable is never used and can be safely removed
Unused argument	(W0613)	Function argument is never used and can be remov.
Redefined outer name	(W0621)	Inner variable name shadows outer name
Too many nested blocks	(R1702)	Has more than five levels of nested blocks
Too complex	(R1260)	Cyclomatic complexity is higher than 10

Table 7.2: Some of the most common error/warning messages in Pylint [89]

The warnings used in this thesis are on Table 7.2. Currently, the cyclomatic complexity setting is optional and must be enabled manually in the command line.

7.3 Evaluation datasets

7.3.1 Boa data science dataset

For the set that consists of data science projects a preexisting dataset will be used.
It is the Boa Dataset of Data Science Software in Python Language by Biswas et al. [64].
This was originally created for the Boa infrastructure by the Iowa State University which is supposed to provide an infrastructure for easier data mining [90].

The dataset as the name suggests only consists of data science projects that are written in Python. Additional criteria were applied from said paper.

One of the criteria is that the repository used a data science related keyword, such as “data-science”, “machine-learning”, “big-data”, etc. It also had to use at least one typical data science related library, such as sklearn, pandas, numpy, tensorflow, theano, etc. which can be seen in Listing 7.3. Finally the GitHub repository had to have a minimum number of 80 stars. One drawback of using keywords is that setting the keyword for a GitHub repository is voluntary, but this is currently the only meaningful way to categorize repositories by a topic. All keywords are seen in Listing 7.3.

1558 repositories were found that matched these criteria. For reproductive purposes, the author wrote a list of all collected repositories which can be used to clone these repositories. However, as this list was written in February 2019, 18 of its entries are missing from GitHub and cannot be cloned anymore. More on that in the subsection 5.3.4 “Filtering the Python code dataset” as the filtering process is similar.

This set will only be used for evaluation purposes, it is separate from the general Python code dataset, neither repository can be in both sets.

```
"machine-learn", "machine-learning", "data-sci", "data-science", "big-
data", "large-data", "data-analy", "data-analysis "deep-learn", "deep-
learning", "data-model", "artificial-intelligence", "mining", "topic
modelling", "topic-modelling", "natural-language-processing", "nlp",
"data-frame", "dataframe", "data-processing", "ml", "tensorflow",
"tensor-flow", "theano", "caffe", "keras", "scikit-learn", "kaggle",
"spark", "hadoop", "mapreduce", "hdfs", "neuralnet", "neural-net"
```

Listing 7.3: Data science keywords used in GitHub search [64]

For the Boa data science dataset [64] the same steps are done in filtering the dataset, otherwise the procedure is the same as in subsection 5.3.4. One additional step is caused by missing repositories or those that are set to private. 18 of these could not be cloned from GitHub for these reasons. This dataset was also not updated to take account for newer data science projects that were released after the creation of the Boa data science dataset. This process is seen on Figure 7.4.

Two repositories were removed due to it being classified as malware, three more were removed for not having any Python files. 505 are removed for not being Python 3 compatible. One more repository was removed for causing problems with Pylint.
This results in 1029 repositories that are left and are used to represent data science projects.

7.3.2 Generic Python code dataset

The same generic Python code dataset from section 5.3 will be used to build the abbreviation tree and build the single-letter variable relation lists and forms the basis for the next dataset. Some results may only be shown in the full tables that are found in the appendix. For more details, see section 5.3.

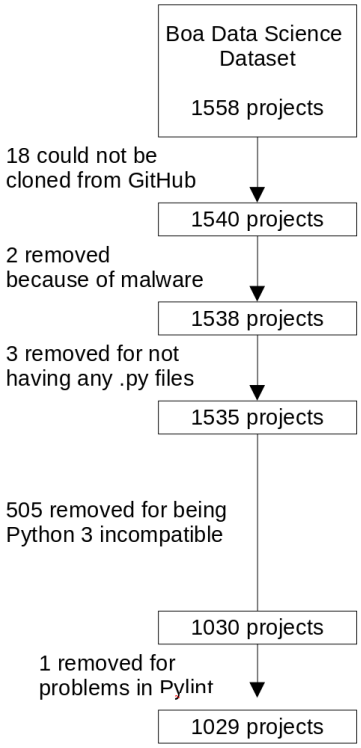


Figure 7.4: Filtering the Boa data science dataset [64]

7.3.3 Non data science dataset

To allow a better comparison, another dataset was derived from the generic Python code dataset which was described in section 5.3.

The difference is that all repositories that contained at least one data science related GitHub keyword in Listing 7.3 was removed. Which are the same keywords that were used to created the Boa data science dataset [64]. This ensures in theory that data science related repositories should not be present in this dataset. This is not guaranteed, as it is possible that a data science project simply does not have any keywords, which are voluntary in GitHub. Using this method the number of repositories for the non data science dataset was reduced from 20925 repositories to 17665 repositories.

Another approach to ensure that no data science projects would be present is to remove any repositories that used certain data science related libraries. Library usage can be detected by looking what libraries are imported in the Python code files. The problem is that data science related libraries have a very broad use and can also be used for non data science purposes. If all repositories that contained any data science related libraries, then more than a half repositories would be removed.

Therefore the decision to remove any repositories with the reason that they contain data science related libraries was not made.

7.3.4 Control dataset

For the last set of evaluations in section 7.8 and section 7.9 a control dataset is used to verify the correctness of the results regarding the non data science dataset. This is separate, using the same criteria as in section 5.3, but with a range of 90 to 93 GitHub stars, resulting into 990 GitHub repositories that are not found in the other three datasets.
All datasets are then cloned to the hard drive, to be processed in the next sections.

7.4 Initial Pylint evaluation

In this section only the number of non-compliant names will be taken into account. Using the command below, Pylint can be applied to all Python files in a single repository:

```
shopt -s globstar;
pylint **/*.py --load-plugins=pylint.extensions.mccabe;
```

Listing 7.5: Command to run Pylint on all Python files recursively on the directory

The shopt [91] command is required to enable the **/* wildcard. Then, all .py files on that directory and all subdirectories will be found and processed by Pylint. At the end a text file will be written which lists all Pylint warnings for that repository. This is done with every repository in a given dataset. The additional option is used to enable scanning for cyclomatic complexity. The entire command is seen in Listing 7.5.

The normalized percentage as written here is the total number of Pylint warnings that show the C0103 warning in Pylint divided by the total number of source lines of code (SLOC). A source line of code in Python is defined as a non-empty line of actual code that are not comments or documentation. The basic methodology is based of a previous work by Simmons et al. [9]. The formula is seen below in Listing 7.6.

$$average = \frac{Pylint\ warnings}{SLOC\ with >= 1\ character}$$

Listing 7.6: Formula for calculating average percentage per repository [9]

The average value refers to the total number of C0103 warnings divided by the total number of SLOC of every single Python file from every single repository in the dataset.

Pylint warning	DS mean	DS median	Non-DS mean	Non-DS median
Non compliant name	5,892%	7,015%	5,180%	5,114%

Table 7.7: Percentage of non compliant name (C0103) warnings. Lower is better

To calculate the median value, the after mentioned command in Listing 7.6 will be run on every individual repository and its percentage is calculated individually. Then insert all values into a single list and sort the list. If the list has an odd number of entries, the value from the middle index will be taken. The middle index of the list is the size of the list minus one, then divided by two. This is always an integer value in odd-sized lists.

In even sized lists, there is no real middle index, applying the above calculation would result in a non-integer value. Therefore the two closest values from the non-integer middle index would be taken, whose index always has a distance of 0.5. These two values would be taken, both of them are added up and then divided by two to obtain the median value of an even-sized list [92].

As seen in Table 7.7, there is a small difference when it comes to non compliant names between the repositories in the Boa Data Science dataset [64] and the other repositories that are in the non-DS dataset. There is a slightly larger difference in the median value. Either way it appears that data science projects have slightly more non-compliant names, relatively speaking, but only by a small margin, that is when a large generic Python code dataset is used as the counterpart. These warnings can be broken down in which case the warning occurred, either in a variable name, an argument name, a constant name or in an attribute name.

The Pylint configuration files were removed in all repositories and is effectively replaced by the default configuration file. There are other types of C0103 errors, such as bad class or function names but these are ignored as they are not used for variables.

It can then be further broken down why the warning occurred, which either occurs if a non-whitelisted variable name has a length of two characters or less or if said length is 30 characters or longer. If neither is the case then it has the wrong naming style, not the one that is recommended by PEP 8 [24]. The results are in Table 7.8.

The variable names i, j, k, ex and _ which have a string length of two or less are whitelisted exemptions in the default configuration files. The first three are typical loop variables as already described in section 3.8. The underscore character is a special variable name, most notably used as a placeholder when a function returns multiple values, but only certain values are used. The other unused ones use the underscore character and are not to be used again in the code. Single-letter homoglyphs already have too short names and are marked as such.

% of reported C0103 errors	DS dataset	Non-DS dataset
Variable name <= 2 characters	55,730%	47,991%
Variable name >= 30 characters	0.008%	0,063%
Not using snake_case naming style	9,144%	10,747%
All C0103 variable warnings	64.881%	58,880%
Argument name <= 2 characters	12,780%	11,828%
Argument name >= 30 characters	0,002%	0,012%
Not using snake_case naming style	2,684%	3,950%
All C0103 argument warnings	15,467%	15,789%
Constant name <= 2 characters	0,168%	0,297%
Constant name >= 30 characters	0,041%	0,101%
Not using ALL_UPPERCASE naming style	5,565%	6,840%
All C0103 constant warnings	5,774%	7,238%
Attribute name <= 2 characters	1,855%	1,752%
Attribute name >= 30 characters	0.002%	0,035%
Not using snake_case naming style	1,284%	3,665%
All C0103 attribute warnings	3,141%	5,452%

Table 7.8: Breakdown of C0103 warnings. Lower is better

Repositories in the data science dataset appears to use very short variable names more frequently than in other non data science based repositories. On the flip side, generic non data science repositories are slightly more likely to use the incorrect the naming style, such as not using the snake_case naming style in variable names. Excessively long variable names are almost non-existent the data science dataset and slightly more prevalent in the non-DS dataset. The result is that the total number of C0103 warnings do somewhat balance each other out, resulting in the numbers seen in Table 7.7. It is possible to have a name that is too short and has an invalid name, which usually occurs with single-letter names that are not whitelisted, such as the capital letter “X”. In that case it is counted as having a too short name, not as using a non-standard naming style.

This entire section does not apply to non-English words that have a string length between 3 and 29 characters and use the correct naming style.

The only notable instance of not using the correct naming style seems to be confined to constants, these are expected to be fully capitalized. One possible explanation is that programmers are unaware to what counts as constant in Pylint, which are all variables that are not inside a function or class.

7.5 Code quality and naming convention compliance

7.5.1 Effects on code quality in general

Pylint warning	DS mean	DS median	Non-DS mean	Non-DS median
Non compliant name	**5,892%**	**7,015%**	**5,180%**	**5,114%**
Missing module docstring	*0,667%*	*0,832%*	*0,647%*	*0,856%*
Missing class docstring	*0,687%*	*0,353%*	*0,780%*	*0,524%*
Missing function docstring	*2,993%*	*2,827%*	*2,807%*	*3,077%*
Line too long	1,769%	2,024%	3,071%	1,923%
Trailing whitespace*	0,843%	0,143%	1,217%	0,099%
*Wrong import order**	*0,446%*	*0,571%*	*0,396%*	*0,447%*
*Method could be function**	*0,446%*	*0,076%*	*0,356%*	*0,087%*
Too many instance attributes	*0,078%*	*0,076%*	*0,113%*	*0,047%*
Too many branches	*0,076%*	*0,014%*	*0,105%*	*0,036%*
Too many arguments	*0,336%*	*0,333%*	*0,285%*	*0,152%*
Too many local variables	*0,281%*	*0,360%*	*0,233%*	*0,150%*
Too complex	*0,127%*	*0,078%*	*0,164%*	*0,110%*
Too many nested blocks	*0,013%*	*0,000%*	*0,029%*	*0,000%*
Bad indentation*	11,244%	0,000%	3,523%	0,000%
Unused variable*	0,284%	0,302%	0,278%	0,202%
Unused argument*	0,413%	0,165%	0,481%	0,154%
Redefined outer name	0,461%	0,150%	0,333%	0,135%

Table 7.9: Percentage of Pylint warnings. Lower is better

The basic methodology is based of a previous work by Simmons et al. [9], but with different datasets. The same calculation formula from the previous section applies here too, but now with a larger variety of Pylint [71] warnings. These metrics from Pylint are chosen because these the some of the most common Pylint warnings that refer to the overall code quality while keeping the table at a reasonable size. The results are seen on Table 7.9.

Certain warnings are less common because these can only occur once per function, per class or per module. Aside from the “missing docstring” warnings, which are self-explanatory, “Too many branches”, “Too many arguments”, “Too many local variables”, and “Method could be function” can only occur once per function. The “Too many instance attributes” warning can only occur once per class. These warnings are italicized in the Table 7.9. Warnings that could be very easily fixed by external tools have a star next to it.

In contrast of the previous analysis by Simmons et al. [9], this non data science dataset not seem to have a significantly better code quality compared to the Boa data science dataset [64]. One possibility is that a much larger number of repositories were used in the non data science dataset. Documentation wise, class docstrings are less frequent in data science projects, but there is no large difference when it comes to functions or modules.

One aspect that is not taken into account regarding documentation are private classes and functions. Typically, by the principle of information hiding [93], private classes and functions are supposed to remain untouched by other programmers except for the one who programmed said private code. Python does not have any actual private classes and functions that are enforced by the Python interpreter. By the PEP 8 convention [24], a single leading underscore character in front of an identifier indicates that something is supposed to be private. As it is not strictly enforced, these private classes and functions can still be changed. Another trait of private classes and functions is that they are typically not documented, as the documentation is meant for public classes and functions which are used by other programmers. Since private functionality is not implemented, this means that Pylint is also not able to tell if it is supposed to be "public" or "private". This applies to both datasets and the differences are small. The warning "Method could be function" also has this issue, some may be unaware of it.

One caveat with the "Line too long" warning is that this warning only set to 100 for some unspecified reason by its developer. PEP 8 [24] recommends a maximum line length of 79, which is very close to the typical recommended maximum line length of 80. In theory, a lower maximum line length causes shorter variable identifiers in order to comply to the maximum line length limit. The maximum line length limit of any code is a rather controversial topic on its own, so this will not be discussed any further. For that matter, non data science projects actually have more cases that have too long lines than data science projects. The same applies to trailing whitespace in code.

The most significant difference is "bad indentation", data science projects have significantly more cases of "bad indentation" than non data science projects. This usually occurs because there are only two spaces instead of the standard four. Eight spaces of indentation were thought to discourage code nesting [94], but this is not the case in data science projects which are less prone to excessive cyclomatic complexity or from having too many nested blocks compared to non-DS projects. DS projects presumably use 2 space indentations. Indentation is a very subjective and controversial subject, in which more spaces are not strictly better, so this will not be discussed any further. More tables can be seen in the appendix section A.2.1. In general the PEP 8 conventions are only guidelines, not hard rules. Therefore, for an example, if a function absolutely needs one hundred arguments and it is completely unavoidable, then the programmer can write it as such. It is meant for cases where it can be avoided and where it is possible to refactor it into something that can comply these code conventions.

Removing any definitive Data Science repositories from the generic code corpus to generate the Non-DS dataset does not significantly change the results for the Non-DS dataset, which can be seen in the appendix, in section A.2.

These results are not comparable to the results that are found in the paper by Simmons et al. [9], because the datasets are different, that paper used a much smaller non-DS dataset. That paper concludes that there is higher number of excessive local variables and arguments. Here there difference is still there, but smaller when assuming the mean value, but not the median value. For the other warning types the differences are too small, except for the "redefined outer name" warning which is slightly more common in DS projects.

7.5.2 Difference in datasets with and without non compliant names

Pylint warning	DS mean	DS median	Non-DS mean	Non-DS median
Non compliant name	**7,967%**	**9,684%**	**7,876%**	**8,134%**
Missing module docstring	*0,427%*	*0,812%*	*0,606%*	*0,843%*
Missing class docstring	*0,596%*	*0,381%*	*0,914%*	*0,569%*
Missing function docstring	*3,439%*	*3,393%*	*3,560%*	*3,761%*
Line too long	2,003%	2,400%	3,478%	2,377%
Trailing whitespace*	0,879%	0,120%	1,663%	0,069%
*Wrong import order**	*0,466%*	*0,613%*	*0,504%*	*0,489%*
*Method could be function**	*0,526%*	*0,063%*	*0,454%*	*0,073%*
Too many instance attributes	*0,091%*	*0,073%*	*0,150%*	*0,048%*
Too many branches	*0,092%*	*0,067%*	*0,144%*	*0,036%*
Too many arguments	*0,385%*	*0,394%*	*0,369%*	*0,176%*
Too many local variables	*0,343%*	*0,440%*	*0,309%*	*0,192%*
Too complex	*0,153%*	*0,088%*	*0,224%*	*0,132%*
Too many nested blocks	*0,017%*	*0,000%*	*0,042%*	*0,000%*
Bad indentation*	12,464%	0,000%	4,610%	0,000%
Unused variable*	0,342%	0,361%	0,387%	0,248%
Unused argument*	0,482%	0,176%	0,599%	0,157%
Redefined outer name	0,565%	0,180%	0,441%	0,138%

Table 7.10: Percentage of warnings in files that contain non compliant names

The same evaluation is done similarly as in the previous subsection but files are only counted if these contained at least one “Non compliant name (C0103)” warning. Basic methodology is loosely based of Simmons et al. [9] as in the previous section. The results are shown on Table 7.10. Table 7.11 that never has non compliant names obviously has zero such results. Another evaluation was done but only with files that did not contain any “Non compliant name (C0103)” warnings. Its results are shown on Table 7.11. For both evaluations, the methodology was otherwise identical to the evaluation in subsection 7.5.1.

The goal is to find out if the code quality is better in code that uses convention compliant names and vice versa. As expected, the code quality is worse when there are non-compliant names and the code quality is better when no non-compliant names are found, because the percentage of warnings is mostly higher across the table when Table 7.10 is compared to Table 7.11.

There are slightly more missing module docstrings in any code that does not have non compliant names, but at the same time, function docstrings are usually more frequent when the variable names use compliant names. Otherwise the metrics are better across the board when naming conventions are followed.

This means that there is that the quality of variable names can be correlated with the overall quality of the code. This answers **[RQ4]**, which asks if there is a relevance between naming conventions and other code features and and there is indeed a correlation of non-complying variable names and code quality. This also applies in reverse, having good variable name correlates to a higher code quality.

Warning/Error	DS mean	DS median	Non-DS mean	Non-DS median
Non compliant name	**0,000%**	**0,000%**	**0,000%**	**0,000%**
Missing module docstring	*1,414%*	*1,159%*	*0,776%*	*1,235%*
Missing class docstring	*0,987%*	*0,000%*	*0,558%*	*0,000%*
Missing function docstring	*1,806%*	*0,824%*	*1,451%*	*0,725%*
Line too long	1,156%	0,000%	2,440%	0,000%
Trailing whitespace*	0,775%	0,000%	0,383%	0,000%
*Wrong import order**	*0,407%*	*0,000%*	*0,201%*	*0,000%*
*Method could be function**	*0,231%*	*0,000%*	*0,179%*	*0,000%*
Too many instance attributes	*0,040%*	*0,000%*	*0,045%*	*0,000%*
Too many branches	*0,032%*	*0,000%*	*0,032%*	*0,000%*
Too many arguments	*0,208%*	*0,000%*	*0,131%*	*0,000%*
Too many local variables	*0,111%*	*0,000%*	*0,092%*	*0,000%*
Too complex	*0,055%*	*0,000%*	*0,053%*	*0,000%*
Too many nested blocks	*0,003%*	*0,000%*	*0,005%*	*0,000%*
Bad indentation*	8,137%	0,000%	1,532%	0,000%
Unused variable*	0,127%	0,000%	0,074%	0,000%
Unused argument*	0,226%	0,000%	0,269%	0,000%
Redefined outer name	0,174%	0,000%	0,134%	0,000%

Table 7.11: Percentage of warnings in files that do not contain non compliant names

However, correlation does not mean causation. In this context, having non-compliant names correlates to worse code quality, but does that also mean that having bad names **will cause** worse code quality?

Just because code that have variable names that violate the PEP 8 naming conventions have a slightly worse quality than code that conforms to the PEP 8 naming conventions, that is not a proof that conforming to the PEP 8 naming convention will automatically lead to a significantly better code quality.

If an ideal tool was used to would automatically fix these identifier names so that they conform the naming conventions, the rest of the code would still not be fixed.

This observation is also averaged across millions of lines of code, so individual cases and outliers cannot be seen here. Good code quality is not just having good identifier names. Having good names do help and it allows for some self-documenting code [20]. This alone is not enough. A high code quality also requires some actual documentation and keeping the individual functions and classes small and clean.

Another argument is that sometimes the use of single-letter identifiers are necessary, but the rest of the code can still be written clean and is well documented.

In short, not conforming to naming conventions correlates, but does not automatically guarantee worse code quality.

However, at the same time, there is no reason not to use compliant names unless they have to be described with a single-letter or to other technical reasons. There are no strict technical reasons to keep the names short in Python. All tables with raw data can be seen in the appendix in section A.2. The median in most of the entries in Table 7.11 is 0%.

7.6 Most popular variable names

This section takes a closer look at the most popular variable names from each dataset. To obtain the variable names, the methodology from section 5.8 is applied which counts the names and then the occurrence of said name is divided by the total number of all occurrences to obtain the percentage value.

Table 7.12 only shows the 35 most popular variable and parameter name declarations from each dataset. A larger table can be found in the appendix tables A.10 to A.12. Because the datasets have a different number of repositories and lines of code, the values are normalized to a relative percentage number. The percentage refers to how frequent a variable name is used, out of one hundred percent. For an example, 9,777% of all variable and parameter name declarations in the data science dataset are named “self”. Every single variable and parameter declaration is count. Inner redefinitions also count. This is supposed to reflect how popular a certain variable name is. To reiterate, every parameter and variable statement counts.

The most popular name in both datasets is “self”, which is very commonly used in classes as a parameter, and is recommended in the PEP 8 [24] coding conventions to be always used as the first parameter in any instance method. Related to this, while far less common, “cls” is recommended to be used as the first parameter in any “static” class methods.

The most notable variable name that violates the existing code conventions is “X”, which violates it in two ways. First, it is too short with a string length of just one. Second, it is capitalized and is used as a variable, not a constant, so it should not be capitalized. At the same time though, it is commonly used as a variable name for matrices. Which does leave a question on why a slightly more descriptive name such as “matrix_X” is not used instead. At a very first glance “X” and “x” look somewhat similar, which could confuse some. Non data science projects seem to use more input and output based variable names, as seen with names such as “response”, “url” and “request”.

In contrast, data science projects have more evidence of certain variable names that indicate the use of certain data science libraries such as “df” and “plotly_name”. This also applies to certain data science related words such as “expected” and “labels”. Single-letter variables such as “y” and “v” are also more common.

One caveat with this evaluation is that these variable names are investigated without any additional context, such as which function or class they are situated at, the actual usage of said variable or any additional comments or documentation that could describe these names in detail for the given context. This concession is necessary because of the large scale of having to process thousands of repositories and to provide a simple list of the most frequently used variable and parameter names. This is a simple popularity ranking. Name categories, such as single-letter variables, are already done in the previous section. Ignoring the datasets and as a side note, there is possibly a tendency to use shorter variable names in Python. The names in some object-oriented programming languages, such as Java, seem to be longer. As there is no comparison with said languages are done and most data science projects are done in Python, this will not be further elaborated.

Variable names that are unique to the DS dataset within Table 7.12 within the Top 35 are underlined. For some reason the word “image” is more popular in the DS dataset than the word “img” in the other datasets. For reference, the word “img” is at the 58th spot in the DS dataset popularity ranking and the word “image” is at the 57th spot in the non-DS dataset.

No		DS	% of all variables		Non-DS	% of all variables
1		self	9,777 %		self	10,364 %
2		x	1,503 %		x	0,993 %
3		data	0,795 %		data	0,722 %
4		_	0,727 %		name	0,665 %
5		result	0,710 %		i	0,661 %
6		y	0,703 %		value	0,653 %
7		v	0,670 %		result	0,594 %
8		name	0,641 %		params	0,521 %
9		i	0,612 %		_	0,438 %
10		model	0,510 %		model	0,366 %
11		expected	0,495 %		response	0,355 %
12		X	0,486 %		args	0,336 %
13		inputs	0,477 %		url	0,323 %
14		val	0,346 %		out	0,300 %
15		config	0,335 %		cls	0,297 %
16		df	0,320 %		path	0,284 %
17		out	0,318 %		config	0,274 %
18		path	0,303 %		key	0,270 %
19		batch_size	0,286 %		y	0,265 %
20		a	0,285 %		s	0,263 %
21		args	0,275 %		request	0,254 %
22		value	0,270 %		a	0,229 %
23		s	0,267 %		output	0,224 %
24		output	0,261 %		res	0,211 %
25		parent_name	0,260 %		p	0,201 %
26		plotly_name	0,260 %		text	0,201 %
27		labels	0,259 %		m	0,200 %
28		params	0,258 %		context	0,200 %
29		dtype	0,255 %		msg	0,197 %
30		cls	0,245 %		c	0,187 %
31		image	0,240 %		r	0,185 %
32		b	0,231 %		b	0,183 %
33		res	0,230 %		state	0,183 %
34		n	0,222 %		img	0,181 %
35		shape	0,209 %		t	0,176 %

Table 7.12: Most popular variable and parameter name declarations for each dataset

To answer the third research question **[RQ3]**, the difference in naming conventions between data science and non data science datasets is that data science projects are more likely to use very short variable names, usually lifted from a mathematical formula.

This includes specific mathematical terms, such as “X”, often used for matrices. Also found in data science projects is a higher chance of using specific naming based on data science specific libraries. On a large scale the differences are less pronounced.

7.7 Abbreviation expansion accuracy

7.7.1 Methodology

Expanding abbreviations as described from section 5.5 to section 5.21 is evaluated here using a small sample of 400 of the most popular abbreviations from each dataset that contain at least three alphabetic characters. The type is ignored for this evaluation. Something somewhat similarly was done in the work by Lawrie et al. [21], but that used a smaller sample of 64 names of which 36 of these are single-letter variables. For the purposes of comparison, it only had a sample size of 28 names that had at least two characters. Its accuracy was 64% if the name had three characters or more. Only the top 1 result was taken into account, not any top 3 or top 5 results as in this evaluation.

The reason why the abbreviation must have three alphabetic characters in this evaluation is to reduce the chance of abbreviations that are too ambiguous, which might be the case with abbreviations that consist of just two alphabetic characters. The goal is mainly to test the accuracy of the abbreviation expansion algorithm. The results are only valid for this case.

For this evaluation, a clear expected answer is required. For an example, it is expected that the abbreviation “params” expands into “parameters” and “config” into “configuration”. This is likely when the abbreviation has three characters or more.

Only in this case, the abbreviation expander returns up to five results, rather than the typical three results. Each of the 400 abbreviations that are evaluated have an expected full word which is set manually. An example output is seen in Table 7.13.

Abbr.	Expected result	1st result	2nd result	3rd result	On?
params	parameters	parameters	parametrizations	prepare_pretrained_models	1st
args	arguments	arguments	argvs	adjust_to_origins	1st
url	uniform_resource_ locator	underlying	upgrade_node_image _version_initial	update_table_throughput_ initial	FAIL
config	configuration	configuration	configurator	configure	1st
msg	message	message	messenger	missing	1st

Table 7.13: 1st 5 results of abbreviation expansion accuracy evaluation, non-DS dataset

The top 1 result would therefore be the best result that would be highlighted in the IDE, while the remaining two results in the top 3 results are not highlighted. If the returned full word from the abbreviation tree data structure matches the expected full word from the evaluation, then it is a correct result. Otherwise it is not. For the Top 2, Top 3, and Top 5 results, if one of the results match the expected result then it is considered to be correct.

There is the possibility that the abbreviation is found in the abbreviation tree data structure but neither of the five results have the correct expected full word. Another aspect to evaluate is if there are any abbreviations that do not show up in the abbreviation tree data structure at all. An abbreviation is only in the tree data structure if it has been resolved at least once in the prior creation of the abbreviation tree data structure

7.7.2 Results

A control dataset which is similar to the non data science dataset and has been explained in section 7.3 is now being used here to verify the results from the non data science datasets. The data from the control dataset was not used to figure out the full words from the abbreviations. More details in that section. Given how similar the results are when compared to the non data science dataset, the results in Table 7.14 appear to be correct.

400 popular names from	Top 1 result	Top 2 results	Top 3 results	Top 5 results	In Abr. Tree
DS dataset	61,00%	68,75%	**71,25%**	71,75%	90,00%
Non-DS dataset	65,75%	74,75%	**77,00%**	77,50%	96,50%
Control dataset	65,25%	74,50%	**76,75%**	77,50%	91,75%

Table 7.14: Accuracy on abbreviation expansion, trained on generic dataset

Unless noted, the abbreviation tree was generated from the generic Python code dataset. The accuracy of the data science dataset when it comes to resolving abbreviations appears to be lower when compared to the non data science dataset. This can be explained because the abbreviation tree was based on the generic Python code dataset, from which the non data science dataset bases on. Some variable names in the data science dataset use really specialized or obscure abbreviations, which are less common and thus harder to resolve, which is why the number of abbreviations found in the tree is also lower. The precision is the accuracy of the top 3 results divided by the number of how many abbreviations can be found in the abbreviation tree data structure at all. For the data science dataset the precision is 79,06%. For the non data science dataset the precision is 80,31%. In comparison, top 1 accuracy of the algorithm from Lawrie et al. [21] was 64% for variable names with at least three characters. The results also justify why three options exist. If only the best option existed, then the effective accuracy is lower. The top 2 results are very close in terms of accuracy to the top 3 results. The top 5 results do not improve the effective accuracy by a large margin. In general, having the option to choose is necessary because there are a few abbreviations which are truly ambiguous and there is no perfect answer for that.

7.7.3 Results with comments and documentation enabled

For one test run on the non data science dataset, the comments and documentation were also taken into account for possible abbreviation expansion. The results are on Table 7.15.

Non-DS Dataset	Top 1 result	Top 2 results	Top 3 results	Top 5 results	In Abr. Tree
Without comments	65,75%	74,75%	**77,00%**	77,50%	96,50%
With comments	60,75%	71,00%	**74,00%**	75,00%	97,25%

Table 7.15 Accuracy on abbreviation expansion with or without comments

When comments are enabled, while more abbreviations are in the abbreviation tree data structure, the accuracy actually decreases. The possible explanation is that there is too many full words to choose from.

Comments may have words that in which abbreviations resolve into correct full words but actually are not the correct and relevant full words. Once there is a large enough dataset, there is no need to include comments. The code from such a large dataset is enough to find the words from the abbreviations, said code also has more relevant words.

7.7.4 Results with a smaller base dataset

What happens if the generic Python code dataset used to generate the abbreviation tree is halved or quartered in the total size? The results are below in Table 7.16.

Non-DS Dataset	Top 1 result	Top 2 results	Top 3 results	Top 5 results	In Abr. Tree
100% generic dataset size	65,75%	74,75%	**77,00%**	77,50%	96,50%
50,0% generic dataset size	59,25%	67,50%	**71,25%**	71,50%	94,25%
25,0% generic dataset size	56,75%	65,50%	**67,50%**	68,25%	89,00%
12,5% generic dataset size	57,00%	62,00%	**64,25%**	65,00%	85,75%

Table 7.16 Acc. on abbreviation expansion, using different datasets for tree generation

As expected, the accuracy from the smaller dataset source that only has 50% as much code as the full generic Python code dataset is lower and supports the idea that a larger dataset will increase the accuracy, but not by much.

However, it is expected that an even larger dataset than the generic Python code dataset will have diminishing returns in accuracy improvement. After all, the accuracy from using the control dataset as source is lower, but not in the single digits, which means that there is no linear improvement based on the size of the dataset. It is expected that using a dataset that is twice as large would only cause small gains in accuracy because this generic Python code dataset is already large enough for this purpose. This was not tested because of technical limitations regarding the hard disk drive space and the time required.

Another aspect that was not explored due to the lack of sufficient data is whenever a data science dataset of the same size of the generic Python code dataset would improve the accuracy in data science specific code. This should be expected to be true but this cannot be proven until such a large data science dataset exists. Regarding **[RQ6]**, more mined data does indeed help with accuracy, at least up to a certain degree. The Top 1 accuracy is however not that much better compared to the algorithm by Lawrie et al. [21] which was at 64% using the top 1 result, which did use additional info such as local documentation. Tailoring towards the data science dataset by using the data science dataset as source for the abbreviation lookup tree does not work because of the small size of the DS dataset, the smaller size causes a lower accuracy. For more details, see the appendix subsection A.2.2.

7.7.5 Reasons of why the accuracy of abbreviation expansion is not perfect

The most problematic abbreviations for the algorithm appear to be acronyms that are supposed to resolve into multiple words. The number of multiple words is equal to the string length of the acronym and every one of these multiple words must start with each letter from the acronym, in correct order. Because of that it can be difficult to find the correct full identifier from the abbreviation. This is especially true if the abbreviation is better known than its fully spelled out name. In some cases, remembering an acronym is easy but remembering the fully spelled out name is not which is why these acronyms are used in the first place. Examples are “url”, “pdf”, and “html” which stand for “Uniform Resource Locator”, “Portable Document Format”, and “HyperText Markup Language” respectively. It might be a better idea to treat such well known acronyms as English words. It also applies to abbreviations that are basically used universally. Examples are “protobuf” and “capsys” which are actually the names used in of some popular libraries. The other case why some of these abbreviations cannot be resolved is because some of the non-English words are not abbreviations at all, but names, such as “alice”, “bert”, and so on.

For obvious reasons these are not abbreviations. Then there are also some words which make no sense at all in the English language and therefore cannot be resolved at all. Examples are words such as “abc”, “xyz”, “foo”, and “baz”.

One set of abbreviations which cannot be resolved by this algorithm are the abbreviations whose full words do not contain all characters from the abbreviation. Such examples are “lineno” which actually stands for “line_number”, the “o” is not present in the latter. Another example is “klass” which is a way to write “class” because “class” is a Python reserved keyword. The former is not a proper English word. Then there is “i18n” or “l10n” which stand for “internationalization” and “localization” respectively. No number is present in the full word.

One limitation caused by the soft word splitter is that the word length must be at least six so it can be separated into two shorter words that have a word length of at least three. This means that if there is an identifier that consists of two words and has a word length of five or less then it cannot be resolved. One example is “mobj” cannot be resolved because the word length is less than six, but this is supposed to resolve into “media_object”.

As this evaluation only focuses on 400 popular abbreviations, the real accuracy when used with obscure abbreviations is expected to be lower.

One last reminder is that all results only apply to abbreviations that have at least three alphabetic characters, not less. This also does not work on single-letter variable names because it is too ambiguous to get a full word from a single letter. One attempt to solve that instead will be explained in the next section.

7.8 Accuracy of single-letter variable replacement

The control dataset is used to the accuracy of the single-letter variable replacement, because it was trained on the generic Python code dataset and the non data science dataset is a subset of that set. The accuracy is basically perfect on the non data science dataset because it was trained on it. More details in section 7.3. From there on, the other dataset is referred to as the test dataset.

The same algorithm for the single-letter variable replacement was performed on a local code to create the lists necessary for the comparison required for the single-letter variable replacement. The local lists and the global lists from the single-letter variable replacement list tree are then compared, which returns up to three potential variable name replacements.

The accuracy is calculated by checking if one of the three variable name replacements match the original variable name. If a match is found, it is a positive result, if not then it is a negative result. To get the accuracy across all files, the number of positive results will be divided by the total number of results, both positive and negative.

This algorithm should be on par with a partial implementation of JSNEAT [22] that is used in JavaScript , therefore the target accuracy to match its partial implementation, which only incorporates the single variable context and task context is 47,3%. This algorithm for Python only takes the function name and its single variable context into account.

The actual measured accuracy is too low to be usable in real life. It was only at 27,17% for the control dataset and 23,77% for the data science dataset which is way below the target accuracy of 47,3%. The results are virtually identical if the comparison was done with the list order in mind or not.

The accuracy is even lower when type inference is turned on, which is as low as 5,23% for the data science dataset. This could possibly be the effect of overfitting, in which the data is too specific and not generic enough. This is clearly not satisfactory. One possible reason will be explored in the next subsection.

7.8.1 Comparing the complexity of a function

One hypothesis is that the average JavaScript function is less complex than the average Python function and less complex functions are easier to work with for the JSNEAT algorithm by Tran et al. [22]. The algorithm only looks within a single function.

There is one key way to calculate the complexity of a function, which is called cyclomatic complexity or the McCabe metric named after its author, T.J. McCabe [16]. It is a graph-theoretic complexity measure in which a single function is illustrated as a control-flow graph. One node represents a continuous processing task until a conditional or loop is met. From there on, branching is necessary in the control flow. In the graph, this is represented as two or more edges originating from that node, and each edge connects towards a new node. In case of a loop, this new node will connect back to its previous node. In case of an if-else statement, it "splits" into two paths, until both statements finish, in which case it "merges" back into one new node. As a graph the formula is $V(G) = E – N + 2P$. E is the number of edges, N is the number of nodes, P is the number of connected components

In other words, the cyclomatic complexity increases when there are more if-else statements and more loops are in that function. If the value is over 10, then it is hard to read. If it is over 20, it becomes really difficult. Over 30 and it is almost impossible.

Another lesser metric to find out how big a function is is the number of statements in a single function. To an even lesser degree, the number of parameters can be taken into account, but the cyclomatic complexity is by far the most important metric for estimating how complex a function is.

To find out these metrics for Python code, the existing linter Pylint [71] is used on the non data science dataset, but with its settings tuned so that it always triggers a warning on every function for having too many statements, too many parameters or having an excessive cyclomatic complexity. The threshold value for each of these setting is set to the lowest possible value, which is either -1 or 0, depending on which was allowed.

Because Pylint [71] obviously does not work on JavaScript code, a different linter was required for JavaScript. JsHint [95] was chosen and the settings are similarly adjusted to always trigger a warning at every function. Now it needs a large JavaScript dataset. For this evaluation, the same dataset as the one used for the JSNEAT [22] evaluation was used, albeit some repositories could not be downloaded anymore. As of this writing of the thesis, this dataset had 11840 JavaScript repositories, which are the highest rated repositories on GitHub. Some repositories were removed since 2019 when that paper was written.

For every function, the cyclomatic complexity, then number of statements and the number of parameters were recorded. To obtain the mean value [92], that number was divided by the total number of function. An alternative mean value was also calculated to verify the actual mean value by removing the highest and lowest values that make up to 0.1% of all values [96]. This is to remove outliers. Finally the median value is calculated as already described in section 7.4.

One note regarding source lines of code in JavaScript, as this language is different than Python. For this evaluation, one source line of code in JavaScript is one correct non-commented line that uses at least one alphanumeric character. Curly braces ("{"), which are used to mark the beginning and end of a function in JavaScript, do not count as a source line of code. Python functions in classes also may have one additional "self" or "cls" parameter, which increases the mean number of parameters in Python.

Also for this evaluation, as previously, any files with more than 10000 lines are excluded and any files that have a single line of more than 1000 characters are excluded. Files that only have one line are excluded too, as this is usually minified code.

Metrics per function	Python mean	Py. trim. mean	Python median	JS mean	JS trim. mean	JS median
Cyclomatic complexity	2,44	2,44	1	2,62	2,62	1
Statements	7,97	7,98	4	4,82	4,83	2
Parameters	2,06	2,06	2	1,09	1,09	1

Table 7.17: Metrics per function. Lower means it is less complex

Table 7.17 shows that the average JavaScript function is less complex than the average Python function. It does not mean that JavaScript programmers write less complex code than Python programmers. To somewhat counteract this table, another metric was measured, namely how many functions there are per source line of code. In JavaScript it is 0,12 functions per SLOC, while in Python it is 0,07 functions per SLOC. In other words, more functions are written in JavaScript per line of code compared to Python.

Interestingly enough though, there is no significant difference in cyclomatic complexity, which is also a metric how many branches there are in a function.

The mean number of parameters per function is higher in Python than in JavaScript. A higher number of parameters means that there are more variables that JSNEAT [22] has to work with. JSNEAT mainly works with statements for a given variable within a single function and with the function name, of which there can only be one. After that, it also takes into account of all other variables in a single function.

Having the code split up into more functions does not mean that the entire code is less complex. It is actually recommend to keep the functions as reasonably small as possible, which in turn lowers the cyclomatic complexity within a function, therefore enabling easier readability and therefore code quality of said function.

Another metric is how many source lines of code a single file has. For Python the mean SLOC for a single file is 117,88 which is higher than the mean SLOC of a single JavaScript file which is 76,39 SLOC. More lines of code are harder to read. For reference, if a file has more than 1000 SLOC, then it is considered to be very difficult to read by humans. The actual number of lines in a file can be higher because SLOC does not include empty lines, comments, and documentation. At the same the JavaScript and Python have significant differences in many ways even when both are dynamically typed, which explains why trying to implement JSNEAT which was originally meant for JavaScript code into Python will not work well. A completely different algorithm may be required to solve that problem of replacing single-letter variable names. From a strictly technical standpoint, a function in JavaScript is a variable, but not in Python, just to name one example.

To answer **[RQ5]**, the first abbreviation expansion algorithm based on the work by Lawrie et al. [21], it works as expected, with some minor room of improvement, such as using a larger dataset and by adding another intermediate step such as an ideal lemmatizer which can handle unigrams to figure out if the word is really a noun or just a verb.

However, the algorithm based on JSNEAT [22] to recover single-letter variable names does work rather poorly. This is explained because the average Python function is more complex than those used in JavaScript from which the algorithm originated from. The abbreviation expansion algorithm could have been used on single-letter variables but the results would be too ambiguous since there is only one letter to begin with.

7.9 Variable name improvement statistics

In this evaluation all variable and parameter names in every Python file from each dataset will be evaluated and be placed in one of six categories seen in the next three tables.

To obtain the variable names, the methodology from section 5.8 is applied which places the variable name in one of the six categories and each occurrence is counted.

The first category are names that only contain English words, either a single word or multiple words. As seen in Table 7.18, this is the largest category by far.

The second category are names that have multiple words and they contain at least one English word. This category has a smaller percentage than the first one in that table.

The third category are names that contain English words and also contain abbreviations that can be resolved. This only applies to Table 7.19 and Table 7.20.

The fourth category are names that contains multiple words and at least one resolvable abbreviation. This also only applies to these two tables.

The fifth category include all single-letter variables and two-letter variables that start with an alphabetic character and ends with a number. These are referred as Pseudo-single-letter variables from there on and both are treated as identically in this evaluation.

The sixth category are any non-English words that have at least two alphabetic letters. Before abbreviation expansion, it also includes any expandable abbreviations which are not expanded yet. Anything that violates the snake_case convention is also here, due to the lack of proper word separation.

To illustrate the difference, two tables exist. Table 7.18 shows all variable and parameter names that are placed in one of the six categories before any abbreviations were expanded and Table 7.19 is similar to the previous table except that any resolvable abbreviations are expanded in that table. Ignoring the differences between both datasets, the difference between the two tables are that names from the second and the sixth category moved to the third and to a lesser degree to the fourth category, respectively. This shows that it can not only expand fully non-English abbreviation, but can also expand multi-word names that have both English and non-English words.

At the same time the percentages of names that only contain English words and single-letter variables have not changed. In the former case, it is obvious, as fully English words cannot be expanded any further with the existing algorithm. In the latter case, the algorithm does not work at all with single-letter variables and therefore these are ignored.

Ignoring abbreviation expansion, data science projects appear to use single-letter and pseudo-single-letter variables more frequently, which matches up with the results from the previous Table 7.8. This is in line with the hypothesis that data science projects use mathematically named variables more frequently, which are names that are directly derived from mathematical equations without giving them a proper English name.

The difference is far lower when it comes to other non-English variable names, which is within less than one percent at most. This also applies to multiple with a mix of English and non-English words.

This means that the data science dataset has fewer variable names that only contain English words, because there are more single-letter variables.

One caveat with Table 7.19 is that it only shows the number of all resolvable names, without factoring in the accuracy if the abbreviation expansion algorithm itself. Using the precision number of 80,31% from subsection 7.8.1, for non-DS dataset, the effective percentage of the resolvable abbreviations decreases and said percentage moves to the non-English word category in the non data science dataset. The precision in the data science dataset is 79,06%. The formulas to calculate the adjusted percentages are written below.

Category	% (DS)	% (Non-DS)
Only English words	60,982%	64,560%
1+ English word in multiple words	14,844%	14,023%
English words and expandable abbreviations	0,000%	0,000%
1+ resolvable abbreviation in multiple words	0,000%	0,000%
Non-English, (Pseudo-)Single-letter var.	9,454%	7,014%
Non-English, at least 2 alphabetic letters	14,720%	14,403%

Table 7.18: Breakdown of variable and parameter names, before abbreviation expansion

Category	% (DS)	% (Non-DS)
Only English words	60,982%	64.560%
1+ English word in multiple words	5,057%	3,724%
English words and expandable abbreviations	20,009%	20,776%
1+ resolvable abbreviation in multiple words	1,820%	1,480%
Non-English, (Pseudo-)Single-letter var.	9,454%	7,014%
Non-English, at least 2 alphabetic letters	2,678%	2,446%

Table 7.19: Breakdown of variable and parameter names, after abbreviation expansion

Category	% (DS)	% (Non-DS)
Only English words	60,982%	64,560%
1+ English word in multiple words	7,106%	5,752%
English words and expandable abbreviations	15,819%	16,688%
1+ resolvable abbreviation in multiple words	1,439%	1,189%
Non-English, (Pseudo-)Single-letter var.	9,454%	7,014%
Non-English, at least 2 alphabetic letters	5,200%	4,800%

Table 7.20: Similar to Table 7.19 but adjusted for abbreviation expansion accuracy

The first formula applies to the 3rd and 4th category. The second formula applies to the 2nd and 6th category. Old percentages refer to table 7.19, newer ones refer to Table 7.20.

$$AdjustedPercentage = precision(newPercentage)$$

Listing 7.21: Formula for calculating adjusted percentage for 3rd and 4th category

$$AdjustedPercentage = oldPercentage - precision(oldPercentage - newPercentage)$$

Listing 7.22: Formula for calculating adjusted percentage for 2nd and 6th category

The formula in Listing 7.21 simply multiplies the value with the precision percentage. The formula in Listing 7.22 instead performs a subtraction with the old percentage and the new percentage first before it is multiplied with the precision percentage. This value is then used as the subtrahend with the old percentage to calculate the adjusted percentage.

This is verified by checking if all percentages in a column sum up to 100%. This is very close, not exactly 100%, because all values are rounded by three digits past the decimal point and therefore there are slight rounding errors.

When compared to the original algorithm by Lawrie et al. [21], which managed to expand about 16% of all variable names and had an accuracy of 64% for variable names with three or more characters, this new algorithm can expand about 22% of all variable names and has an accuracy of 65% with three or more characters, which is a small improvement. This does not factor in any (pseudo-)single-letter variables which would be potentially about 7% of the recoverable variable names and potentially another 3,5% of existing partial variable names. The accuracy was not factored in both accuracy numbers regarding the number of expanded names. That other paper did not use any Python code in its smaller dataset.

To answer **[RQ1]**, data science projects appear use more single-letter variables than non science Python projects, which violates the default Pylint settings, unless it is whitelisted by the user. These are also harder to recover from compared to abbreviations. This is also confirmed in the findings of Table 7.8, in which the data science projects have more single-letter variable names but also have slightly less naming convention violations compared to non data science projects due to a wrong naming style, relatively speaking. Very long names are virtually non existent in both datasets and are therefore not the culprit. Data science projects also use fewer variables with full English words than non data science projects, also from a relative standpoint, but the difference is not very stark.

7.10 Time performance

Time performance refers to the amount of time taken to perform a single task, which varies depending on the computer. This section is divided into multiple subsections not just for structural reason but also because the key limitation is different for each subsection. Unless noted, most steps only have to be done once, except for the step in subsection 7.10.6. All evaluations are done with the computer described in section 7.1, which has a SSD, unless noted. Time in Java programs is measured with VisualVM [97] which is a Java profiler.

7.10.1 GitHub repository retrieval

To create the generic Python code dataset from section 5.3, after generating the scripts to clone every repository that meets the criteria that are described from said section, the repositories have to be cloned.

$$downloadTime\ s = \frac{repositoriesSizeAs\ MB}{downloadSpeed\ \frac{MB}{s}}$$

Listing 7.23: Formula to calculate ideal download time

The ideal time as seen on Listing 7.23 to clone these repositories is equal to the total size of all repositories in megabytes divided by the user download speed that is expressed as megabytes per second. Most download speeds are expressed with megabits per seconds, which means that the raw number must be divided by eight to replace megabits to megabytes. For an example, 500000 MB worth of GitHub repositories divided by 6,25 MB/s, which is equivalent to 50 Mb/s, takes about 80000 seconds or 22 hours and 13 minutes. In practice however, there are additional limitations. The first limitation is a soft rate limitation by GitHub which occurs when too many repositories are cloned within a short period of time. If the soft rate limit is reached, then the download speed is throttled, until a certain time passes, usually within minutes and the soft rate limit it not violated anymore. The exact threshold when it comes to cloning GitHub repositories is not known.

This is likely when downloading many repositories in bulk, especially if the individual repositories are small, therefore many small repositories are downloaded very quickly, which triggers the soft rate limit.

There is also presumably a hard rate limit, as with every other content provider. If a hard rate limit is reached, then the IP address which sends the requests to clone the repositories would be temporarily banned and therefore be unable to clone anything. It has not been reached so far, but it there should be one, probably only reserved for extreme cases, like Denial of Service attacks or for severe Terms-Of-Service violations.

One more limitation, if an old cloning script is used, that contains repositories which are no longer hosted on GitHub, is that if the user tries to clone a repositories that is either set to private by the author or is no longer available on GitHub, then the command line prompts for the user name and password. This by itself requires user interaction and cannot be automated. No actual time is lost if the user can respond to the prompt quickly, but this also means that it is not possible to run said cloning script unsupervised.

A limitation could occur if the user download speed is very fast but the hard drive is slow. In that case, the write speed of the hard drive becomes the limitation if it is slower than the download speed.

7.10.2 Filtering the dataset

If an anti-virus scanner is used to detect and remove any malicious repositories, then this will take up the majority of the time for filtering the dataset. The only widely known anti-virus scanner on Linux which is ClamAV [98], which only uses one thread and can take up to 15 hours, 33 minutes and 42 seconds to scan the entire Python code dataset for malware. This process could be multi-threaded by running one thread for every GitHub repository, that time could then be divided by the number of threads that the computer can run because these threads can run independently. No information between these threads are passed.

One more notable filtering step that takes up a lot of time is checking for Python 3 compatibility, which is done by checking every Python file with the Python 3 interpreter. This process takes up 1 hour, 4 minutes and 7 seconds and this is already multi-threaded.

The last filtering step that takes a long time to complete is the process of running Pylint [71] across all Python files. Even when it is multi-threaded, this is a very long process which is also very memory intensive. It takes 14 hours, 8 minutes and 3 seconds to complete.

The other steps, which includes checking if there are repositories with no Python files and checking if one repository is already in the Boa Data Science Dataset [64] take up very little time, relatively speaking compared to the other steps. These two steps only require two to three seconds to complete, which is nothing compared to the other steps which may require more than one day to complete.

7.10.3 Type inference

Type inference on the generic Python code dataset can be done independently, before any of the other actions. All results are stored on JSON files. These can be opened instead when type inference is called again, as the majority of the time in type inference is spent on the algorithm itself, not by reading JSON files.

Performing this on the generic Python code dataset takes 30 hours, 36 minutes, 13 seconds using one CPU thread. This process can be multi-threaded in theory, but doing so will significantly increase system memory usage. As such, only four threads were run.

7.10.4 Abbreviation tree data structure generation

It takes 1 hour, 10 minutes and 49 seconds on the SSD to generate the abbreviation tree data structure with multi-threading enabled.

The majority of time was taken up by a single huge Python file that has 196611 lines of code that needed to be processed. Processing on a single file cannot be multi-threaded. The other portion was taken up by sorting and exporting the final abbreviation tree which also cannot be multi-threaded. Due to inconsistency issues with multi-threading, it had to be disabled, which turns it into 1 hour, 26 minutes and 6 seconds.

This does not include the extra overhead caused by the Type4Py type inference algorithm [50] which is described in the previous subsection, because that process can be done separately and the results of the type inference which are JSON files can be stored for later usage so that that algorithm does not have to run again. Most of the was spent processing a single file and exporting the tree data structure which could not be multi-threaded without affecting the file integrity.

If the tree size was reduced, by not including any tree entries with a low priority of five or lower, the total time to generate and export the abbreviation tree data structure would be reduced to 9 minutes and 30 seconds, which is more than one hour of time saved, at a probably lower accuracy.

As a side note, this was also tested on a hard disk drive to verify whenever the slower performance of the hard drive would affect the time required to generate and export this tree. The time required on the hard disk drive to generate the abbreviation tree data structure is 2 hours, 4 minutes and 3 seconds, which does indeed confirm that a hard drive is slower than a solid state drive but at least it is not so slow to the point when it becomes unusable. Another notable point is that multi-threading had to be disabled when using hard drive. One hypothesis is that a hard drive cannot handle as many I/O requests for the small .py files as a SSD. It has to be emphasized that this process only needs to be done once.

7.10.5 Single-letter variable replacement operations

To create a full set of resources for the single-letter variable replacement, it takes 38 minutes and 47 seconds to create it. It takes almost exactly the same amount of time to create the cache which is 37 minutes and 16 seconds, as that routine is very similar to the previous one, just with some filtering. To export the full set of resources it takes 6 minutes and 36 seconds, as it is only a file transfer operation.

7.10.6 Time performance in plugin

Whenever the plugin is used to replace a variable name, the UI thread of the IDE will be blocked. For that reason it has to perform the action very quickly, in two seconds or less, so that the plugin still feels usable enough by the user. This is never an issue in practice because the plugin never requires more than a second to find a solution to replace a variable name under normal usage conditions. It helps that it is mostly a look up based approach, little actual computation is done at the plugin. It does take about one to two seconds to replace a single-letter variable, because slightly more computation is involved.

Note that not every possible variable, file and hardware was tested, so the experience may be worse in a slower computer. It is expected that if the entire project is located in an old hard disk drive instead of a new solid state drive, that the time required to process the variable will take seconds, which may be deemed as unresponsive by some users.

The time required to initially generate all files necessary for the plugin takes more than 83 hours or about three and a half days, which is a rather long time, but this only needs to be done once. Once these files are generated, these files never have to be generated again for at least a decade, unless some major shift in human language occurs. These pre-generated files can then be distributed to every user of the plugin so the user will not have to generate these files by themselves.

To answer the second research question **[RQ2]**, this can be done in a barely reasonable amount of time provided that a SSD is used. Most of the time-consuming parts are caused by external programs, which cannot be optimized unless an entirely new program is written from scratch, which is difficult and time-consuming in terms of development. Internet bandwidth is also a factor. It can only run on Linux, not on any FAT32 or NTFS-based file system because of the restrictions regarding the file name rules, which is less restrictive on ext4, which only runs on Linux. Once everything is created, the time performance in the plugin is fast, about a second or less.

7.11 Threats to validity

This section is mostly based on the previous research paper by Simmons et al. [9], as these threats of validity are similar in this paper and in that paper in regards to evaluation.

7.11.1 Threats to internal validity

Some percentages as seen on Table 7.9 are lower than others because they only count at most once per function or class, rather being able to count at every possible line. Such lower percentages are already italicized on the table. Dividing the numbers by the number of source lines of code is done for compatibility reasons, as it is not possible to keep the number of bad indentations which can occur at any line, below 100%.

Some code files in both datasets contain duplicate code, as reported by Pylint [71]. This would also duplicate the other Pylint warnings if said code would cause Pylint warnings. One explanation is that code can be copied from different resources, like from StackOverflow. One big source of duplicate code are Python virtual environments, as some repositories contain these. In an attempt to eliminate this threat, the algorithm ignores all folders that are named “venv”, which indicates that there is a virtual environment in this folder. Excessively large Python files with more than 10000 lines, which are likely not created from humans are excluded in this evaluation, to avoid outliers that were not made by humans. Empty files are also ignored and would not have altered the results anyway.

The own installation will affect some results on Pylint, such as the installed libraries on the computer, which affect import error messages. For this evaluation, as many libraries were installed on the computer as possible, the main source is the stable Debian package list [99]. This is in contrast to the policy in Simmons et al. [9], which only installed as few Python libraries as possible. This is to minimize any warnings caused by missing imports. The warnings chosen in the chapter should not be affected by the local installations of any Python libraries.

Another caveat is that Pylint configuration files can be found in a very small number of repositories, which alters Pylint behavior. While any comments and configuration files can be easily removed, there is no guarantee that it was everything regarding any flags that ignores Pylint settings.

There are also the chance of false positives. The GitHub page on Pylint [71] reports 152 open issues and 167 closed issues regarding false positives as of May 1st 2022. This should not be a problem if these false positives happen at a constant rate across all 17665 repositories, then is normalized by the large number of source lines of code. The same applies to false negatives, of which there are 34 open and 24 closed issues on its GitHub page as of May 1st 2022.

Despite the best efforts, such as removing all projects that are found in the data science dataset and removing all repositories that use one data science related keyword, there is the possibility that the non data science dataset can still contain some data science projects. One reason is that adding a topic in GitHub is voluntary. The other issue is that sometimes in non data science projects that some data science library is used for other purposes. The large majority of projects in the data science dataset are data science repositories as curated by Biswas et al. [64]. The same applies in reverse for the non data science dataset, as any data science repositories are removed from the non data science dataset. This should still allow to show a relative difference between these two datasets.

7.11.2 Threats to external validity

Like in the paper by Simmons et al. [9], GitHub currently only hosts open source repositories. This means that no proprietary code is included. Obtaining such code on a large scale that is required in the previous evaluation will be very difficult without the approval of every owner that owns said proprietary code and said code may contain trade secrets which the owners will never disclose.

Chapter 8

Conclusion

This final chapter briefly summarizes this thesis, what could be done in the future and the final conclusion of this thesis

8.1 Summary

A plugin for the PyCharm [82] integrated development environment was developed to either expand existing abbreviated variable names into more meaningful ones or replace single-letter variables. It has two different algorithms depending on if it is a single letter or not. All variables except those with a single letter are expanded by using a lookup tree structure which is created by data mining 20925 GitHub repositories that use Python code to find lines that contain abbreviated variable names and their corresponding full word. For single-letter variables which are too ambiguous for abbreviation expansion, a different algorithm is proposed which replaces the name by data mining these same repositories to find the context on how a variable name interacts with other ones based on the statements found inside its function. The algorithms are based off existing ones [21][22].

The evaluation initially tests the claim that data science projects have a slightly worse code quality than regular non data science projects. It was found that in terms of code quality that the data science projects have a more similar code quality when compared to non data science projects, slightly winning and slightly losing out in some aspects which evens out and the differences are small. The biggest difference were in the number of spaces used for indentation, in which data science projects use a non-standard number of spaces such as two spaces, which does not lead to a higher number of nested blocks or higher cyclomatic complexity. There is also a proven correlation between naming convention compliance and overall code quality, but this is not causative. Data science projects are more likely to use data science specific names.

The accuracy of the new abbreviation expansion algorithm is at 65,75% for the top 1 result and 77% if the expected word is in the top 3 results, which is a minor improvement over the 64% accuracy of the old algorithm. For data science projects, the accuracy is at 61% for the top 1 result and 71,25% for the top 3 results. This accuracy increases the larger the dataset is to create the lookup tree structure. However, the accuracy when it comes to replacing single-letter variables is much lower, at 27,17% for the control dataset and 23,77% for the data science dataset, which is too low to be usable. Explanations are that Python still has many differences compared to JavaScript even when both are dynamically typed and because the average JavaScript function is shorter and less complex than the average Python function.

The plugin itself, after a very lengthy dataset creation process or using pre-generated datasets, can resolve abbreviations very quickly.

8.2 Outlook

The user interface on the plugin could be improved, the full name could be shown by simply hovering the mouse over the abbreviation. To take it one step further, hovering the mouse over the abbreviation over a slightly longer period would show a full dictionary definition of the full word from which the abbreviation derives. This should only be an option as this could potentially cause too much visual clutter on the graphical user interface, depending on what definition the user wants to look up. It also assumes that the accuracy is higher since it must rely on the top 1 result.

The plugin currently only works on JetBrains PyCharm [82]. There are many other Python IDEs that could benefit from such a plugin, but they work different enough to the point where a plugin for a different IDE almost needs to be written from scratch, as the GUI elements cannot be ported over and because some internal functionality from PyCharm is used. Only some internal algorithms can be carried over to a plugin that is for a different IDE. Further optimizations can be done to use less hardware resources, allowing older computers to run this plugin.

While the plugin works in both Linux and Windows, the creation of the datasets require Linux. While Linux is free, it would be desirable if said creation would also work on Windows because Windows is by far the most popular operating system for end-users. Future work can also be done for potential future platforms, such as the new line of Apple macOS systems that use a different architecture.

As the number of repositories on GitHub increase over time, a newer and much larger dataset could be created in the future once these repositories are created. A new dataset in general is useful in the future, not just for the larger size which can increase accuracy, but also deal with a potential language shift, which can occur quickly in this fast-living world of technology. Speaking of languages, the abbreviation expansion algorithm does not rely on Python specific features and can be applied universally to other different programming languages, including ones that might be created in the future.

Alternatively, datasets could be created to specifically target certain fields, such as data science, engineering, microelectronics, applications and so on.

The abbreviation expansion algorithm have room for improvement. Currently, aside from using a larger dataset, it could utilize an ideal lemmatizer that does not exist yet. That lemmatizer would be able to tell if a single word is a noun or a verb. Current lemmatizers and part-of-speech taggers rely on full sentences to figure out the context of a single word. However, most of the time variable names are written as single words on their own, not as part of a sentence. Such a lemmatizer would allow to prioritize nouns over verbs, which would lead to a more accurate result. The abbreviation expansion algorithm could be combined with another algorithm to improve the accuracy.

But the single-letter variable replacer algorithm has a low accuracy that it cannot be improved on to become viable. This is because while this approach may work for JavaScript with shorter functions, this does not work in the Python language.

Alternatively, an entirely new algorithm could be invented that could surpass both previously mentioned algorithms in terms on accuracy, but that algorithm would rely on new advanced ideas, such as using deep learning or other advanced and complicated machine learning algorithms.

8.3 Conclusion

The abbreviation expansion algorithm is only a minor improvement compared to the idea by Lawrie et al. [21], which does have some potential for improvement. The single-letter variable replacer, which is based off the idea of Tran et al. [22], when applied to Python code, performed worse than expected. The user plugin works reasonably enough, but it may need some additional optimization so it can run on older computers. The initial creation of the resources takes a long time, but this only needs to be done once.

From a standpoint, it is still an improvement, because a certain type of accuracy is always better than having an accuracy of 0%, because there was no previous attempt at improving variable names in Python, which means it wins by default, so to speak. Or in other words, anything is better than nothing, strictly speaking in a scientific sense. It can also fix bad naming styles easily because naming styles mostly matter regarding how to capitalize individual words and how to indicate the separation of two words in a name.

However something that works badly from af user experience standpoint should not be released at all. The average user has high expectations and therefore expects a plugin that, to put it simply, works. And by works it means, in this context, requires an algorithm with a high to very high accuracy for the first result.

The main goal in the grand scheme is to improve the code quality, which can be accomplished in many ways, including improving the variable names. However, improving said names only goes so far. Good names are less useful if the rest of the code remains difficult to read. This program can only improve these variable names and not improve the rest of the code or add any additional documentation. Self-documenting [20] code, which is achieved by using meaningful names and a clear structure will not be sufficient enough in more complex functions, which in some instances cannot be entirely avoided. Actual documentation in case of Python is still necessary to create any documentation pages for any API which is meant to be used by developers without having to expose any complicated code. In the code, the code can also be commented. There are works that demonstrate automatic documentation generation [18].

Refactoring the function into smaller functions which are easier to read because smaller functions have less lines to read also improves the overall readability of the code. This lowers the cyclomatic complexity which means the code is less complex [16]. This also applies to some degree by splitting a single large code file into multiple smaller ones. These are more likely to follow the-single responsibility principle

One other aspect is consistency. Consistency as in, a common terminology that is agreed across all developers in a team and code conventions that everyone agrees on, such as the maximum characters per line, the number of spaces per indentation, the maximum lines of code per function and of course, the naming conventions. This is relatively easy to agree on in a single team and if this remains a closed source project which no one else reads. Not so much if this project is open source, which can be read by anyone. While PEP 8 [24] tries to set a standard regarding a convention, it does appear that data science projects use a similar but a different enough convention. This can cause friction to new developers who are used to the PEP 8 conventions if they see code that uses different conventions. All Python projects, regardless of field, should utilize a universal set of conventions, such as PEP 8, to minimize the friction of transitioning from one project to another. And use documentation wherever possible.

One more caveat is that the coding style of Python is significantly different to other high-level programming languages, such as Java, which is more verbose than Python. This is also reflected in the variable names, which are usually longer than the ones used in Python. The differences in how code is typically written in a programming language matters, as the JSNEAT [22] that is applied to small functions in JavaScript has difficulties with more complex functions found in Python.

Improving variable names only improves one aspect of code quality, out of many.

Appendix

The appendix are additional instructions which are used if the user wants to create the resources required for the plugin by themselves instead of using the existing pre-generated resources. The second section of the appendix are additional tables that could not fit in the evaluation chapter.

A.1 Additional instructions

A.1.1 Creation of abbreviation tree data structure text file.

Before the program to create and export the abbreviation tree data structure can be started, the docker instance of the Type4Py local model [50] needs to be started. The first command in Listing A.1. only needs to be used once.

```
sudo docker pull ghcr.io/saltudelft/type4py:latest;
sudo docker run -d -p 5001:5010 -it
ghcr.io/saltudelft/type4py:latest
```

Listing A.1: Command to start Type4Py local docker image [50]

Note that this will require at least temporary superuser privileges. It is possible to launch multiple Docker instances by incrementing first port number by one each time another Docker instance is launched, which is 5001 by default. It is not recommended because this will greatly increase system memory usage. No more than a single instance should be launched if the user only has 16GB of RAM and no more than four instances should be launched if the user only has 32GB of RAM.

```
java -jar VariableExtracter.jar --generateAbbreviationMap
[DICTIONARY TEXT FILE] [PYTHON DATA CORPUS ROOT DIRECTORY]
[OUTPUT ABBREVIATION TREE FILE LOCATION]
```

Listing A.2: Command to start the abbreviation tree data structure generation

To start the creation of the abbreviation the program named “VariableExtractor.jar” must be provided with six arguments. All arguments require read privileges.

The first argument “--generateAbbreviationMap” states that the abbreviation tree data structure is created. The remaining parameters are the locations of the files.

The second argument is the absolute file path of the English dictionary text file that contains a list of English words separated by newlines.

The third argument is the root directory of the generic Python data corpus, more details in section 5.2.

The fourth argument is the location of the output file. The user must have write privileges on that location. During the process multiple files will be generated for internal processing and caching, but they can be safely ignored except for the final output file.

A.1.2 Creation of variable relation lists

Three separate commands from the same application are required to create the variable relation list data structure used for single-letter variable name replacement.

All commands will require the -Xmx27000M Java Virtual Machine argument, which allocates 27000 MB of system memory to the Java Virtual Machine. Therefore at least 32GB of system memory is required with nothing else running in the background. If a smaller Python data corpus is used then the Xmx number can be decreased, if a larger corpus is used then more memory is required. The application is named “App.jar”.

A.1.3 Creation of the variable relation list cache folder

Frequently accessed variable relation lists which are large in size should be stored in a cache folder which is then loaded into system memory when the client starts.

```
java -Xmx27000M -jar editor.jar --createCacheFolder
[DICTIONARY TEXT FILE]
[OUTPUT CACHE GRAPH FOLDER LOCATION]
[PYTHON DATA CORPUS ROOT DIRECTORY]
```

Listing A.3: Command to start the creation of the variable relation list cache folder

The first argument is “--createCacheFolder”. This starts the command to create folder that contains the most frequently used variable relation lists.
The second argument is the dictionary text file as used in the previous listings.
The third argument is the exported abbreviation tree as text file as created in section 6.3.
The fourth argument is the output folder location of the most used graphs.
The fifth argument is the location of the Python data corpus as described in section 5.2.

The resulting files are large text files which can be loaded in about a minute into system memory, but are not suitable to access individual small graphs from the hard disk drive. Using the Python data corpus as described in section 5.2, it will create 80 text files that take up 2.4 GB in size.

A.1.4 Creation of the variable relation list full folder

In preparation to export the variable relation list to the hard disk drive, all graphs must be processed in a similar way as in the previous subsection, but now includes all graphs, not just the most frequently used ones.

```
java -Xmx27000M -jar editor.jar --createFullFolder
[DICTIONARY TEXT FILE]
[OUTPUT FULL GRAPH FOLDER LOCATION]
[PYTHON DATA CORPUS ROOT DIRECTORY]
```

Listing A.4: Command to start the creation of the full variable relation list folder

The arguments are identical to Listing 6.5 except the first argument is now called “--createFullFolder” and the output full graph folder location should be different from the output cache graph folder location. It creates 291 text files that take up 7.5 GB in size.

A.1.5 Export the variable relation list tree data structure

The full folder architecture is not suitable in its own form because it consists of a small number of large files. To speed up access times of less frequently used small graphs, the large text files are split up into several smaller ones.

```
java -Xmx27000M -jar editor.jar --export
[FULL GRAPH FOLDER LOCATION] [OUTPUT CACHE FOLDER LOCATION]
```

Listing A.5: Command to export the variable relation list tree data structure

The --export command takes the full graph folder as the second argument as input and the third argument is the cache folder location as the output. Beware that thousands of directories and files are created in a tree-like fashion, which allows fast access if only a single file needs to be accessed but accessing all of them will take a very long time. If the user ever wants to look into these directories it is strongly recommended to use a command line interface, not the standard GUI file explorer.

Using the Python data corpus as described in section 5.2, the first level consists of 1310 folders, the second level consists of 237049 folders with a total of about 12 million files. Because of the very large number of small files, it takes about 69 GB of actual disk space on an ext4 based file system. It could potentially break other older file systems due to be very large number of small files or require a very large amount of hard disk space if the minimum file block size is large.

The Python code corpus is no longer required after the creation of these prerequisites and can be safely deleted to save hard disk drive space.

A.2 Additional tables

A.2.1: Pylint warning tables

Here are more raw Pylint warning tables that could not reasonably fit in the main thesis and are not cleaned up for presentation purposes.

Not only does it include the DS dataset and the non-DS dataset but also the generic Python code dataset. As a reminder, the non-DS dataset was derived from the generic Python code dataset but with all repositories removed that had one data science related GitHub keyword.

This also includes the median value, which was left out in the original tables because Table A.3 mostly had 0% values as median values. It also added unnecessary clutter in the proper tables, so it was moved here for reference. Because of that, the percentage differences between the DS and non-DS dataset cannot be shown here because these would not fit. The tables can be seen in the next two pages.

Pylint warning	DS mean	DS median	Generic mean	Generic median	Non-DS mean	Non-DS median
Non compliant name	**5,892%**	**7,015%**	**5,223%**	**5,286%**	**5,180%**	**5,114%**
Missing module docstring	*0,667%*	*0,832%*	*0,659%*	*0,868%*	*0,647%*	*0,856%*
Missing class docstring	*0,687%*	*0,353%*	*0,758%*	*0,524%*	*0,780%*	*0,524%*
Missing function docstring	*2,993%*	*2,827%*	*2,804%*	*3,071%*	*2,807%*	*3,077%*
Line too long	1,769%	2,024%	3,181%	2,062%	3,071%	1,923%
Trailing whitespace*	0,843%	0,143%	1,172%	0,108%	1,217%	0,099%
*Wrong import order**	*0,446%*	*0,571%*	*0,413%*	*0,467%*	*0,396%*	*0,447%*
*Method could be function**	*0,446%*	*0,076%*	*0,351%*	*0,092%*	*0,356%*	*0,087%*
Too many instance attributes	*0,078%*	*0,076%*	*0,117%*	*0,059%*	*0,113%*	*0,047%*
Too many branches	*0,076%*	*0,014%*	*0,104%*	*0,038%*	*0,105%*	*0,036%*
Too many arguments	*0,336%*	*0,333%*	*0,314%*	*0,188%*	*0,285%*	*0,152%*
Too many local variables	*0,281%*	*0,360%*	*0,252%*	*0,186%*	*0,233%*	*0,150%*
Too complex	*0,127%*	*0,078%*	*0,162%*	*0,111%*	*0,164%*	*0,110%*
Too many nested blocks	*0,013%*	*0,000%*	*0,028%*	*0,000%*	*0,029%*	*0,000%*
Bad indentation*	11,244%	0,000%	3,793%	0,000%	3,523%	0,000%
Unused variable*	0,284%	0,302%	0,286%	0,221%	0,278%	0,202%
Unused argument*	0,413%	0,165%	0,482%	0,160%	0,481%	0,154%
Redefined outer name	0,461%	0,150%	0,349%	0,146%	0,333%	0,135%

Table A.6: Percentage of warnings. Lower is better

Pylint warning	DS mean	DS median	Generic mean	Generic median	Non-DS mean	Non-DS median
Non compliant name	**7,967%**	**9,684%**	**7,535%**	**8,055%**	**7,876%**	**8,134%**
Missing module docstring	*0,427%*	*0,812%*	*0,588%*	*0,826%*	*0,606%*	*0,843%*
Missing class docstring	*0,596%*	*0,381%*	*0,836%*	*0,552%*	*0,914%*	*0,569%*
Missing function docstring	*3,439%*	*3,393%*	*3,349%*	*3,630%*	*3,560%*	*3,761%*
Line too long	2,003%	2,400%	3,429%	2,444%	3,478%	2,377%
Trailing whitespace*	0,879%	0,120%	1,514%	0,078%	1,663%	0,069%
*Wrong import order**	*0,466%*	*0,613%*	*0,496%*	*0,500%*	*0,504%*	*0,489%*
*Method could be function**	*0,526%*	*0,063%*	*0,422%*	*0,078%*	*0,454%*	*0,073%*
Too many instance attributes	*0,091%*	*0,073%*	*0,146%*	*0,062%*	*0,150%*	*0,048%*
Too many branches	*0,092%*	*0,067%*	*0,135%*	*0,038%*	*0,144%*	*0,036%*
Too many arguments	*0,385%*	*0,394%*	*0,384%*	*0,214%*	*0,369%*	*0,176%*
Too many local variables	*0,343%*	*0,440%*	*0,319%*	*0,228%*	*0,309%*	*0,192%*
Too complex	*0,153%*	*0,088%*	*0,209%*	*0,129%*	*0,224%*	*0,132%*
Too many nested blocks	*0,017%*	*0,000%*	*0,038%*	*0,000%*	*0,042%*	*0,000%*
Bad indentation*	12,464%	0,000%	4,684%	0,000%	4,610%	0,000%
Unused variable*	0,342%	0,361%	0,376%	0,260%	0,387%	0,248%
Unused argument*	0,482%	0,176%	0,553%	0,160%	0,599%	0,157%
Redefined outer name	0,565%	0,180%	0,439%	0,148%	0,441%	0,138%

Table A.7: Percentage of warnings in files that do contain compliant names

Warning/Error	DS mean	DS median	Generic mean	Generic median	Non-DS mean	Non-DS median
Non compliant name	**0,000%**	**0,000%**	**0,000%**	**0,000%**	**0,000%**	**0,000%**
Missing module docstring	*1,414%*	*1,159%*	*0,878%*	*1,384%*	*0,776%*	*1,235%*
Missing class docstring	*0,987%*	*0,000%*	*0,624%*	*0,000%*	*0,558%*	*0,000%*
Missing function docstring	*1,806%*	*0,824%*	*1,686%*	*0,919%*	*1,451%*	*0,725%*
Line too long	1,156%	0,000%	2,810%	0,000%	2,440%	0,000%
Trailing whitespace*	0,775%	0,000%	0,432%	0,000%	0,383%	0,000%
Wrong import order*	*0,407%*	*0,000%*	*0,242%*	*0,000%*	*0,201%*	*0,000%*
Method could be function*	*0,231%*	*0,000%*	*0,203%*	*0,000%*	*0,179%*	*0,000%*
Too many instance attributes	*0,040%*	*0,000%*	*0,055%*	*0,000%*	*0,045%*	*0,000%*
Too many branches	*0,032%*	*0,000%*	*0,036%*	*0,000%*	*0,032%*	*0,000%*
Too many arguments	*0,208%*	*0,000%*	*0,167%*	*0,000%*	*0,131%*	*0,000%*
Too many local variables	*0,111%*	*0,000%*	*0,109%*	*0,000%*	*0,092%*	*0,000%*
Too complex	*0,055%*	*0,000%*	*0,059%*	*0,000%*	*0,053%*	*0,000%*
Too many nested blocks	*0,003%*	*0,000%*	*0,005%*	*0,000%*	*0,005%*	*0,000%*
Bad indentation*	8,137%	0,000%	1,908%	0,000%	1,532%	0,000%
Unused variable*	0,127%	0,000%	0,090%	0,000%	0,074%	0,000%
Unused argument*	0,226%	0,000%	0,346%	0,000%	0,269%	0,000%
Redefined outer name	0,174%	0,000%	0,155%	0,000%	0,134%	0,000%

Table A.8: Percentage of warnings in files that do not contain non compliant names.

A.2.2: Additional table regarding abbreviation expansion accuracy

One note regarding Table A.9, the main difference is that instead of using the much larger generic Python code dataset the data science dataset was used to create the abbreviation lookup tree data structure. Then it was applied to the most popular abbreviations that had more than three characters from the data science dataset. The idea here is to test if tailoring it towards data science will improve the accuracy regarding abbreviations used in data science projects. In practice, because the data science dataset is only about 5% of the size of the full generic dataset, the accuracy could not be improved by using a smaller data science dataset. A comparably large dataset of data science projects with high quality code does not exist yet.

When compared to the similarly sized control dataset, which is similar to non data science projects, the data science dataset as source of the abbreviation lookup tree wins out against using the control dataset as that source, meaning that tailoring towards data science projects does work, but a larger dataset will still win out.

On DS Dataset	Top 1 result	Top 2 results	Top 3 results	Top 5 results	In Abr. Tree
Generic set as source	59,50%	67,50%	**70,00%**	70,50%	88,25%
Control set as source	38,75%	47,25%	**48,50%**	49,00%	67,25%
DS set as source	50,75%	55,75%	**57,50%**	58,00%	77,50%

Table A.9 Accuracy on abbreviation expansion with or without comments

A.2.3: Most popular variable and parameter name declarations

To fit all of the top 100 most popular variable and parameter name declarations, it had to be spread across three tables, which would not fit in the main thesis.

No	DS	% of all vars.	Generic	% of all vars.	Non-DS	% of all vars.
1	self	9,777 %	self	9,957 %	self	10,364 %
2	x	1,503 %	x	1,123 %	x	0,993 %
3	data	0,795 %	data	0,700 %	data	0,722 %
4	_	0,727 %	i	0,662 %	name	0,665 %
5	result	0,710 %	name	0,636 %	i	0,661 %
6	y	0,703 %	value	0,590 %	value	0,653 %
7	v	0,670 %	result	0,557 %	result	0,594 %
8	name	0,641 %	_	0,482 %	params	0,521 %
9	i	0,612 %	params	0,470 %	_	0,438 %
10	model	0,510 %	model	0,432 %	model	0,366 %
11	expected	0,495 %	args	0,346 %	response	0,355 %
12	X	0,486 %	y	0,316 %	args	0,336 %
13	inputs	0,477 %	out	0,316 %	url	0,323 %
14	val	0,346 %	config	0,310 %	out	0,300 %
15	config	0,335 %	response	0,304 %	cls	0,297 %
16	df	0,320 %	cls	0,287 %	path	0,284 %
17	out	0,318 %	url	0,277 %	config	0,274 %
18	path	0,303 %	path	0,272 %	key	0,270 %
19	batch_size	0,286 %	key	0,255 %	y	0,265 %
20	a	0,285 %	s	0,251 %	s	0,263 %
21	args	0,275 %	output	0,246 %	request	0,254 %
22	value	0,270 %	a	0,225 %	a	0,229 %
23	s	0,267 %	request	0,214 %	output	0,224 %
24	output	0,261 %	res	0,204 %	res	0,211 %
25	parent_name	0,260 %	p	0,204 %	p	0,201 %
26	plotly_name	0,260 %	text	0,203 %	text	0,201 %
27	labels	0,259 %	img	0,190 %	m	0,200 %
28	params	0,258 %	state	0,188 %	context	0,200 %
29	dtype	0,255 %	m	0,188 %	msg	0,197 %
30	cls	0,245 %	b	0,181 %	c	0,187 %
31	image	0,240 %	c	0,179 %	r	0,185 %
32	b	0,231 %	context	0,178 %	b	0,183 %
33	res	0,230 %	n	0,177 %	state	0,183 %
34	n	0,222 %	r	0,176 %	img	0,181 %
35	shape	0,209 %	t	0,176 %	t	0,176 %

Table A.10: Most popular variable and parameter name declarations for each dataset.

As with the previous tables, the generic Python code dataset was added here too, as a reference.

No	DS	% of all vars.	Generic	% of all vars.	Non-DS	% of all vars.
36	loss	0,209 %	msg	0,174 %	n	0,175 %
37	key	0,206 %	loss	0,173 %	d	0,175 %
38	index	0,203 %	d	0,170 %	index	0,167 %
39	outputs	0,200 %	index	0,170 %	results	0,166 %
40	f	0,194 %	inputs	0,163 %	client	0,162 %
41	msg	0,192 %	line	0,156 %	line	0,160 %
42	filename	0,186 %	f	0,156 %	f	0,157 %
43	net	0,185 %	results	0,155 %	filename	0,153 %
44	t	0,179 %	labels	0,155 %	k	0,150 %
45	h	0,177 %	k	0,153 %	kwargs	0,149 %
46	node	0,175 %	filename	0,150 %	v	0,147 %
47	text	0,174 %	image	0,148 %	other	0,146 %
48	d	0,171 %	h	0,147 %	item	0,142 %
49	values	0,166 %	batch_size	0,146 %	h	0,142 %
50	c	0,165 %	df	0,145 %	loss	0,141 %
51	k	0,163 %	parser	0,143 %	message	0,140 %
52	mask	0,160 %	v	0,143 %	target	0,139 %
53	dataset	0,157 %	logger	0,143 %	obj	0,137 %
54	p	0,153 %	client	0,142 %	parser	0,136 %
55	w	0,152 %	w	0,142 %	logger	0,136 %
56	kwargs	0,149 %	target	0,141 %	w	0,135 %
57	idx	0,147 %	kwargs	0,140 %	image	0,134 %
58	img	0,137 %	mask	0,136 %	ret	0,131 %
59	arg	0,137 %	outputs	0,132 %	labels	0,131 %
60	line	0,135 %	item	0,128 %	node	0,129 %
61	m	0,132 %	other	0,126 %	expected	0,128 %
62	layer	0,131 %	ret	0,126 %	size	0,125 %
63	parser	0,131 %	expected	0,125 %	version	0,124 %
64	target	0,130 %	node	0,124 %	id	0,124 %
65	weights	0,129 %	obj	0,124 %	mask	0,123 %
66	size	0,126 %	message	0,124 %	inputs	0,121 %
67	g	0,125 %	size	0,122 %	metadata	0,116 %
68	features	0,124 %	idx	0,119 %	df	0,114 %
69	z	0,119 %	device	0,118 %	query_parameters	0,114 %
70	func	0,117 %	dataset	0,117 %	batch_size	0,113 %

Table A.11: Most popular variable and parameter name declarations for each dataset.

No	DS	% of all vars.	Generic	% of all vars.	Non-DS	% of all vars.
71	state	0,116 %	label	0,116 %	idx	0,113 %
72	r	0,115 %	version	0,114 %	type	0,111 %
73	label	0,113 %	j	0,112 %	j	0,110 %
74	axis	0,112 %	id	0,110 %	header_parameters	0,110 %
75	width	0,112 %	X	0,108 %	val	0,108 %
76	optimizer	0,111 %	metadata	0,104 %	label	0,108 %
77	other	0,111 %	all	0,104 %	return_value	0,108 %
78	all	0,111 %	val	0,102 %	outputs	0,105 %
79	batch	0,111 %	input	0,101 %	device	0,105 %
80	mod	0,110 %	start	0,100 %	all	0,103 %
81	device	0,110 %	optimizer	0,099 %	info	0,103 %
82	url	0,107 %	type	0,098 %	headers	0,102 %
83	metadata	0,107 %	return_value	0,097 %	ctx	0,102 %
84	results	0,105 %	count	0,095 %	event	0,100 %
85	logits	0,105 %	info	0,094 %	count	0,100 %
86	ret	0,104 %	ctx	0,094 %	start	0,100 %
87	obj	0,104 %	mode	0,094 %	status	0,099 %
88	input_shape	0,103 %	query_parameters	0,094 %	user	0,099 %
89	op	0,103 %	stride	0,092 %	input	0,095 %
90	start	0,103 %	header_parameters	0,090 %	dataset	0,094 %
91	fields_by_name	0,101 %	width	0,089 %	mode	0,091 %
92	indices	0,101 %	headers	0,087 %	width	0,089 %
93	logger	0,100 %	status	0,086 %	query	0,088 %
94	j	0,098 %	event	0,086 %	row	0,087 %
95	mode	0,098 %	user	0,084 %	stride	0,087 %
96	x_train	0,097 %	features	0,083 %	cfg	0,086 %
97	height	0,095 %	values	0,082 %	cmd	0,086 %
98	images	0,093 %	query	0,082 %	optimizer	0,085 %
99	scores	0,090 %	cfg	0,082 %	title	0,084 %
100	step	0,090 %	env	0,081 %	options	0,084 %

Table A.12: Most popular variable and parameter name declarations for each dataset.

A.2.4: Example output of abbreviation expansion accuracy evaluation

The tables on the following pages shows a small sample of the output of the abbreviation accuracy evaluation. Due having 400 tests and each table only fitting 30 results, it would take 14 pages for each of the four datasets, up to 56 pages in total, therefore only a small portion is shown for exemplary purposes. The tables start with the most popular abbreviation, followed by slightly less popular abbreviations after each entry, in order.

As per section 7.8, they do not contain any variable names that have less than two alphabetic characters, see that section for more details.

Abbr.	Expected result	1st result	2nd result	3rd result	On?
params	parameters	parameters	parametrizations	prepare_pretrained_models	1st
args	arguments	arguments	argvs	adjust_to_origins	1st
url	uniform_resource_ locator	underlying	upgrade_node_image _version_initial	update_table_throughput_ initial	FAIL
config	configuration	configuration	configurator	configure	1st
msg	message	message	messenger	missing	1st
img	image	image	inverse_perspective_ mapping	imagesize	1st
obj	object	object	objective	objection	1st
ret	ret	result	request	return	1st
idx	index	index	indexe	indexer	1st
val	value	value	variable	validator	1st
info	information	information	inform	identification	1st
ctx	context	context	create_index	core_text	1st
cfg	configuration	configuration	control_flow_graph	classification_model_for_ testing	1st
cmd	command	command	commander	camera_script_after_render	1st
str	string	string	strip	stream	1st
env	environment	environment	environ	envelope	1st
func	function	function	functool	forward_function	1st
dtype	data_type	data_type	datatype	device_type_dummy	1st
resp	response	response	relationship	register_post	1st
opt	option	optimizer	option	optimization	2nd
loc	location	location	locator	local	1st
param	parameter	parameter	parametrization	prepare_pretrained_model	1st
pred	prediction	predict	prediction	predicate	2nd
udf_ params	user_defined_function _parameters	user_defined_ function_ parameters	(no 2nd result)	(no 3rd result)	1st
dim	dimension	dimension	dimensionality	days_in_month	1st
doc	document	document	docstring	documentation	1st
col	column	column	color	collection	1st
src	source	source	search	surface	1st
err	error	error	extract_response	electra_correct	1st
arg	argument	argument	argvs	adjust_to_origin	1st

Table A.13: 1st 30 results of the abbreviation expansion accuracy evaluation, non-DS d.set

“self” and “cls” are special variable names set in the PEP 8 [24] convention and are therefore not listed here.

Abbr.	Expected result	1st result	2nd result	3rd result	On?
val	value	value	variable	validator	1st
config	configuration	configuration	configurator	configure	1st
args	arguments	arguments	argvs	adjust_to_origins	1st
params	parameters	parameters	parametrizations	prepare_pretrained_models	1st
dtype	data_type	data_type	datatype	device_type_dummy	1st
msg	message	message	messenger	missing	1st
idx	index	index	indexe	indexer	1st
img	image	image	inverse_perspective_mapping	imagesize	1st
arg	argument	argument	argv	adjust_to_origin	1st
func	function	function	functool	forward_function	1st
url	uniform_resource_locator	underlying	upgrade_node_image_version_initial	update_table_throughput_initial	FAIL
obj	object	object	objective	objection	1st
ret	ret	result	request	return	1st
expr	expression	expression	experiment	experiment_expression	1st
y_pred	y_prediction	ys_predict	y1_predict	y_predict	FAIL
rng	random_number_generator	range	random_tensor_generator	random_number_generator	3rd
attrs	attributes	attributes	array_to_float_buffers	attributions	1st
doc	document	document	docstring	documentation	1st
var	variable	variable	variance	variant	1st
allowlist	allow_list	allow_list	(no 2nd result)	(no 3rd result)	1st
env	environment	environment	environ	envelope	1st
ctx	context	context	create_index	core_text	1st
sess	session	session	start_transform_session	start_session_and_load_model	1st
hparams	hyperparameters	horizontal_pod_auto_scaler_rams	(no 2nd result)	(no 3rd result)	FAIL
fig	figure	figure	fitting	freshfig	1st
info	information	information	inform	identification	1st
exp	expression	experiment	expression	explained	2nd
aug	augmentation	augmentation	augment	augmentor	1st
vocab	vocabulary	vocabulary	vocabularie	vocabcompoiler	1st
pred	prediction	predict	prediction	predicate	2nd

Table A.14: 1st results of the abbreviation expansion accuracy evaluation, DS dataset

Abbr.	Expected result	1st result	2nd result	3rd result	On?
config	configuration	configuration	configurator	configure	1st
args	arguments	arguments	argvs	adjust_to_origins	1st
params	parameters	parameters	parametrizations	prepare_pretrained_models	1st
msg	message	message	messenger	missing	1st
img	image	image	inverse_perspective_mapping	imagesize	1st
ret	ret	result	request	return	1st
idx	index	index	indexe	indexer	1st
url	uniform_resource_locator	underlying	upgrade_node_image_version_initial	update_table_throughput_initial	FAIL
val	value	value	variable	validator	1st
obj	object	object	objective	objection	1st
ctx	context	context	create_index	core_text	1st
dtype	data_type	data_type	datatype	device_type_dummy	1st
env	environment	environment	environ	envelope	1st
doc	document	document	docstring	documentation	1st
func	function	function	functool	forward_function	1st
info	information	information	inform	identification	1st
cmd	command	command	commander	camera_script_after_render	1st
cfg	configuration	configuration	control_flow_graph	classification_model_for_testing	1st
resp	response	response	relationship	register_post	1st
str	string	string	strip	stream	1st
col	column	column	color	collection	1st
pred	prediction	predict	prediction	predicate	2nd
param	parameter	parameter	parametrization	prepare_pretrained_model	1st
pkt	packet	packet	(no 2nd result)	(no 3rd result)	1st
var	variable	variable	variance	variant	1st
fname	file_name	filename	file_name	first_name	2nd
opt	option	optimizer	option	optimization	2nd
arg	argument	argument	argvs	adjust_to_origin	1st
tmp	temporary	temperature	temporary	template	2nd
exp	expression	experiment	expression	explained	2nd

Table A.15: 1st 30 results of the abbreviation expansion accuracy evaluation, control dataset

A.3 Contents on disc

The contents of the discs are briefly mentioned under the list

- The full digital thesis is saved as a .pdf file right on the root directory of the disc.
- The source .odt file of the thesis is located in the folder “thesis_source”.
- All compressed plugin resources and the plugin file itself are in the directory “0_PluginResources”.
- All shell scripts required to clone the datasets are in the directory “0_CloningScripts”. Each script must be pasted in their own separate directory.
- “0_OtherResources” was another generated resource but is not required.
- Older projects are in “stubsIgnoreThem”. These should be ignored and are no longer required.
- Additional instructions in a text file named “readme.txt”.
- “.git” is a hidden directory for Git purposes and can also be ignored.
- The remaining nine directories are the Java programs that are still utilized. Each directory contains a corresponding .jar file which is used to run the program. It is strongly recommend to not move the .jar file as some resources for the programs are located inside these directories.
- Some of these directories contain the original results from the original evaluation, these must not be mixed up with newer generated results.

Not included are the datasets because when uncompressed they take up more than one terabyte of space. There is no way to compress that to fit it on a single DVD.

Note that when cloning repositories, there is a chance that the author of the repository may have deleted it, which means it becomes inaccessible. This will slightly alter the results of any future evaluation.

Pylint [71] and JsHint [95] must be installed separately. They must be installed in a way that it is accessible from the bash command line, such as using “sudo apt install pylint” or the equivalent package manager.

Type4Py [50] is also not included. Exact instructions on how to obtain its local model are in the “readme.txt” file. It should be possible to run everything without type inference in order to speed up the resource creation process but this may alter any evaluation results.

Cache files which take up a lot of space are also not included. They are not required for the initial run of the program and will be generated as needed for future runs. This also means that the first run of the program will take a lot more time than subsequent ones.

For decompressing the files, the program 7-Zip [100] is required. This can also be a lengthy process in terms of time.

Bibliography

[1] Deighan, D. (2016, June 17). Why Data Science is Important & How It's Helping Business. *LinkedIn*. https://www.linkedin.com/pulse/why-data-science-important-how-its-helping-business-damien-deighan

[2] Schroeder, B. (2021, June 11). The Data Analytics Profession And Employment Is Exploding—Three Trends That Matter. *Forbes*. https://www.forbes.com/sites/bernhardschroeder/2021/06/11/the-data-analytics-profession-and-employment-is-exploding-three-trends-that-matter/

[3] Davenport, T. H., & Patil, D. J. (2012). Data scientist. *Harvard business review*, *90*(5), 70-76.

[4] GitHub (n.d.) Search – topic:data-science – GitHub. *GitHub.* Retrieved November 30, 2021, https://github.com/search?q=topic%3Adata-science&type=repositories

[5] Perkel, J. M. (2018). Why Jupyter is data scientists' computational notebook of choice. *Nature*, *563*(7732), 145-147.

[6] TIOBE (2021, November). TIOBE Index for November 2021. *TIOBE*. Retrieved November 30, 2021, https://www.tiobe.com/tiobe-index/

[7] Brewster (2021, May). *16 Examples of Global Companies Using Python in 2021.* Trio. Retrieved November 30, 2021, https://trio.dev/blog/companies-using-python

[8] Koehrsen, W. (2019, July 8). Data Scientists: Your Variable Names are Awful. Here's How to Fix Them. *Towards Data Science.* https://towardsdatascience.com/data-scientists-your-variable-names-are-awful-heres-how-to-fix-them 89053d2855be

[9] Simmons, A. J., Barnett, S., Rivera-Villicana, J., Bajaj, A., & Vasa, R. (2020, October). A large-scale comparative analysis of Coding Standard conformance in pen-Source Data Science projects. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1-11).

[10] Caprile, B., & Tonella, P. (2000, October). Restructuring Program Identifier Names. In *icsm* (pp. 97-107).

[11] Deissenboeck, F., & Pizka, M. (2006). Concise and consistent naming. *Software Quality Journal*, *14*(3), 261-282.

[12] Lawrie, D., Feild, H., & Binkley, D. (2006, September). Syntactic identifier conciseness and consistency. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation* (pp. 139-148). IEEE.

[13] Lyman, I. (2021, October 18). Code quality: a concern for busiensses, bottom lines and empathetic programmers. *The Overflow. https://stackoverflow.blog/2021/10/18/code-quality-a-concern-for-businesses-bottom-lines-and-empathetic-programmers/*

[14] Whiteney, L. (2019, December 17). Software developers say they feel pressure to sacrifice code quality to meet deadlines. *TechRepublic*. https://www.techrepublic.com/article/software-developers-say-they-feel-pressure-to-sacrifice-code-quality-to-meet-deadlines/

[15] Pantiuchina, J., Lanza, M., & Bavota, G. (2018, September). Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 80-91). IEEE.

[16] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4), 308-320.

[17] W. P. Stevens, G. J. Myers, and L. L. Constantine, “Structured design,” IBM Syst. J., pp. 115–139, 1974

[18] Zhai, J., Huang, J., Ma, S., Zhang, X., Tan, L., Zhao, J., & Qin, F. (2016, May). Automatic model generation from documentation for Java API functions. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (pp. 380-391). IEEE.

[19] Bloch, M., Blumberg, S., & Laartz, J. (2012). Delivering large-scale IT projects on time, on budget, and on value. Harvard Business Review, 2-7.

[20] Nielsen, S., & Tollemark, D. (2016). Code readability: Code comments OR self-documenting code: How does the choice affect the readability of the code?.

[21] Lawrie, D., Feild, H., & Binkley, D. (2007, September). Extracting meaning from abbreviated identifiers. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)* (pp. 213-222). IEEE.

[22] Tran, H., Tran, N., Nguyen, S., Nguyen, H., & Nguyen, T. N. (2019, May). Recovering variable names for minified code with usage contexts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (pp. 1165-1175). IEEE.

[23] Pamula, H. (2022, March 22). Cylinder Volume Calculator. https://www.omnicalculator.com/math/cylinder-volume

[24] Van Rossum, G., Warsaw, B., & Coghlan, N. (2001). PEP 8: style guide for Python code. *Python. Org*, *1565*.

[25] w3resource (2020, Feburary 26). C Variables and Constants. w3resource https://www.w3resource.com/c-programming/c-variable.php

[26] Adams, J. C., Brainerd, W. S., Martin, J. T., Smith, B. T., & Wagener, J. L. (1992). *fortran 90 Handbook* (Vol. 32), 2-3. New York: McGraw-Hill.

[27] Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

[28] Van Rossum, G., & Drake Jr, F. L. (1995). *Python tutorial* (Vol. 620). Amsterdam: Centrum voor Wiskunde en Informatica.

[29] van Emde Boas, P., Kaas, R., & Zijlstra, E. (1976). Design and implementation of an efficient priority queue. *Mathematical systems theory*, *10*(1), 99-127.

[30] Chazelle, B. (2000). The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM (JACM)*, *47*(6), 1012-1027.

[31] Gunawardena A. (2008). Tree data structure. https://www.cs.cmu.edu/~clo/www/CMU/DataStructures/Lessons/lesson4_1.htm

[32] Quinlan, J. R. (1996). Learning decision tree classifiers. *ACM Computing Surveys (CSUR)*, *28*(1), 71-72.

[33] Opyrchal, L., & Prakash, A. (1999, June). Efficient object serialization in Java. In *Proceedings. 19th IEEE International Conference on Distributed Computing*

[34] Kettler R. (2012). A Guide to Python's Magic Methods. *GitHub*. https://rszalski.github.io/magicmethods/

[35] Woodbridge, J., Anderson, H. S., Ahuja, A., & Grant, D. (2018, May). Detecting homoglyph attacks with a siamese neural network. In *2018 IEEE Security and Privacy Workshops (SPW)* (pp. 22-28). IEEE.

[36] Robson, A. (2018). Fonts for Displaying Program Code in LATEX.

[37] Hofmeister, J., Siegmund, J., & Holt, D. V. (2017, February). Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)* (pp. 217-227). IEEE.

[38] Cambridge University Press (n.d). Definition of abbreviation. In *Cambridge Advanced Learner's Dictionary & Thesaurus.* Retrieved March 31, 2022 from https://dictionary.cambridge.org/dictionary/english/abbreviation

[39] Cambridge University Press (n.d). Definition of bat. In *Cambridge Advanced Learner's Dictionary & Thesaurus.* Retrieved March 31, 2022 from https://dictionary.cambridge.org/dictionary/english/bat

[40] Hargrove P. H., Whitlock S. T., Boman E. (1997). Fortran 77 Tutorial: Variables, types and declarations. *Stanford University.* https://web.stanford.edu/class/me200c/tutorial_77/05_variables.html

[41] Cajori, F. (1993). *A history of mathematical notations* (Vol. 2), 61.

[42] Cajori, F. (1993). *A history of mathematical notations* (Vol. 1), 381-383.

[43] Jivani, A. G. (2011). A comparative study of stemming algorithms. *Int. J. Comp. Tech. Appl*, *2*(6), 1930-1938.

[44] Van Rijsbergen, C. J., Robertson, S. E., & Porter, M. F. (1980). *New models in probabilistic information retrieval* (Vol. 5587). London: British Library Research and Development Department.

[45] OpenNLP, A. (2011). Apache Software Foundation. http://opennlp.apache.org

[46] Bird, S. G., & Loper, E. (2004). NLTK: the natural language toolkit. Association for Computational Linguistics.

[47] Van Rossum, G., Lehtosalo, J., & Langa, L. (2014). PEP 484–type hints. *Index of Python Enhancement Proposals*.

[48] Bailey, C (n.d.). Pros and Cons of Type Hints. Retrieve February 17, 2022 from https://realpython.com/lessons/pros-and-cons-type-hints/

[49] Lehtosalo, J., Rossum, G. V., Levkivskyi, I., Sullivan, M. J., Fisher, D., Price, G., ... & Zijlstra, J. (2021). Mypy: Optional Static Typing for Python.

[50] Mir, A. M., Latoskinas, E., Proksch, S., & Gousios, G. (2021). Type4py: Deep similarity learning-based type inference for python. *arXiv preprint arXiv:2101.04470*.

[51] D. Jurafsky and J. H. Martin, Speech and Language Processing (2nd Edition). USA: Prentice-Hall, Inc., 2009.

[52] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[53] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, Deep learning. MIT press Cambridge, 2016, vol. 1.

[54] S. Chopra, R. Hadsell, and Y. LeCun, “Learning a similarity metric discriminatively, with application to face verification,” in 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05), vol. 1. IEEE, 2005, pp. 539–546

[55] Allamanis, M., Barr, E. T., Ducousso, S., & Gao, Z. (2020, June). Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation* (pp. 91-105).

[56] Einar L., Liam N. (n.d.). Online JavaScript Beautifier (v1.14.0). Retrieved January 25, 2022 from https://beautifier.io/

[57] Raychev, V., Vechev, M., & Krause, A. (2015). Predicting program properties from "big code". *ACM SIGPLAN Notices*, *50*(1), 111-124.

[58] Vasilescu, B., Casalnuovo, C., & Devanbu, P. (2017, August). Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (pp. 683-693).

[59] Lacomis, J., Yin, P., Schwartz, E., Allamanis, M., Le Goues, C., Neubig, G., & Vasilescu, B. (2019, November). DIRE: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 628-639). IEEE.

[60] Chen, Q., Lacomis, J., Schwartz, E. J., Le Goues, C., Neubig, G., & Vasilescu, B. (2021). Augmenting Decompiler Output with Learned Variable Names and Types. *arXiv preprint arXiv:2108.06363*.

[61] Lawrie, D., Morrell, C., Feild, H., & Binkley, D. (2006, June). What's in a Name? A Study of Identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)* (pp. 3-12). IEEE.

[62] Schankin, A., Berger, A., Holt, D. V., Hofmeister, J. C., Riedel, T., & Beigl, M. (2018, May). Descriptive compound identifier names improve source code comprehension. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)* (pp. 31-3109). IEEE.

[63] Binkley, D., Lawrie, D., Maex, S., & Morrell, C. (2009). Identifier length and limited programmer memory. *Science of Computer Programming*, *74*(7), 430-445.

[64] Biswas, S., Islam, M. J., Huang, Y., & Rajan, H. (2019, May). Boa meets python: A boa dataset of data science software in python language. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (pp. 577-581). IEEE.

[65] Church, D. (2012, January 15). Wictionary top 100,000 most frequently-used English words [for john the ripper]. *GitHub Gist* https://gist.github.com/h3xx/1976236

[66] GitHub, Inc. (2022). About – GitHub. Retrieved February 17, 2022, from https://github.com/about

[67] GitHub, Inc. (n.d.). Search – GitHub Docs. Retrieved November 24, 2021, from https://docs.github.com/en/rest/reference/search

[68] ytisf (2021, September 13). theZoo: A repository of LIVE malwares for your own joy and pleasure. theZoo is a project created to make the possibility of malware analysis open and available to the public. *GitHub*. https://github.com/ytisf/theZoo

[69] Python Software Foundation (2020, January 1). Sunsetting Python 2. https://www.python.org/doc/sunset-python-2/

[70] Raschka, S. (2014, June 1). *The key differences between Python 2.7.x and Python 3.x with examples.* https://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html#xrange

[71] Python Code Quality Authority (2022, April 6). Pylint: It's not just a linter that annoys you! *GitHub.* https://github.com/PyCQA/pylint

[72] Goodger, D., & van Rossum, G. (2001). PEP 257–Docstring conventions. *Documentation, Python Software Foundation*.

[73] W3Schools (n.d.). Python keywords. Retrieved April 25, 2022 from https://www.w3schools.com/python/python_ref_keywords.asp

[74] W3Schools (n.d.). Python assignment operators. Retrieved April 25, 2022 from https://www.w3schools.com/python/gloss_python_assignment_operators.asp

[75] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., & Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE transactions on software engineering*, *28*(10), 970-983.

[76] Python Software Foundation (n.d.). Coroutines and Tasks. Retrieved April 25, 2022 from https://docs.python.org/3/library/asyncio-task.html

[77] *The reverse process of stemming* (2012, February 29). StackOverflow. https://stackoverflow.com/questions/9481081/the-reverse-process-of-stemming

[78] Grammarly (2020, December 16). Plural Nouns: Rules and Examples. *Grammarly* https://www.grammarly.com/blog/plural-nouns/

[79] Webster, J. J., & Kit, C. (1992). Tokenization as the initial phase in NLP. In *COLING 1992 Volume 4: The 14th International Conference on Computational Linguistics*.

[80] Brill, E. (1992). *A simple rule-based part of speech tagger*. PENNSYLVANIA UNIV PHILADELPHIA DEPT OF COMPUTER AND INFORMATION SCIENCE.

[81] VanRossum, G., & Drake, F. L. (2010). *The python language reference* Amsterdam, Netherlands: Python Software Foundation, 7.

[82] JetBrains, S. R. O. (2017). PyCharm. *https://www. jetbrains. com/pycharm/*

[83] JetBrains, S. R. O. (n.d.). Intellij Platform Plugin Template: Template repository for creating plugins for IntelliJ Platform. *GitHub*. Retrieved April 4, 2022 from https://github.com/JetBrains/intellij-platform-plugin-template

[84] Friedman, M. B. (2003). Virtual memory constraints in 32-bit Windows. In *Int. CMG Conference* (pp. 45-56).

[85] JetBrains, S. R. O. (n.d.). JetBrains Marketplace. Retrieved April 4 2022 from https://plugins.jetbrains.com/

[86] Rodeh, O., Bacik, J., & Mason, C. (2013). BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, *9*(3), 1-32.

[87] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., & Peck, G. (1996, January). Scalability in the XFS File System. In *USENIX Annual Technical Conference* (Vol. 15).

[88] Rodeh, O., & Teperman, A. (2003, April). zFS-a scalable distributed file system using object disks. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings.* (pp. 207-218). IEEE.

[89] Logilab (n.d.). pylint(1) – Linux man page. Retrieved April 8, 2022 from https://linux.die.net/man/1/pylint

[90] Iowa State University (2015). *About the Boa Language and Infrastructure.* http://boa.cs.iastate.edu/index.php

[91] Free Software Foundation (n.d.). The Shopt Builtin. Retrieved April 22, 2022 from https://www.gnu.org/software/bash/manual/html_node/The-Shopt-Builtin.html

[92] Cherry, K. (2020, March 24). How to Identify and Calculate the Mean, Median and Mode. https://www.verywellmind.com/how-to-identify-and-calculate-the-mean-median-or-mode-2795785

[93] Parnas, D. L. (1972). A technique for software module specification with examples. *Communications of the ACM*, *15*(5), 330-336.

[94] L. Torvalds (2016). Linux kernel coding style. https://www.kernel.org/doc/html/v4.10/process/coding-style.html

[95] R. Waldron, C. Potter, M. Pennisi, L. Page (2022). JSHint https://jshint.com/about/

[96] W. Kenton, S. Anderson, H.D. Jasperson (2021, July 17). Trimmed Mean. *Investopedia.* https://www.investopedia.com/terms/t/trimmed_mean.asp

[97] Sedlacek, J., & Hurka, T. (2017). VisualVM All-in-One Java Troubleshooting Tool. https://visualvm.github.io

[98] Cisco Systems (n.d.). Clam Antivirus (ClamAV). Retrieved April 8, 2022, from https://www.clamav.net/about

[99] SPI Inc. (n.d.). Debian stable package lists. Retrieved April 18, 2022 from https://packages.debian.org/stable/

[100] Pavlov, I.: 7-Zip (n.d.). Retrieved May 2, 2022 from https://www.7-zip.org