An empirical study investigating the use of Python identifier names

Siqi Tong

A thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science in Computer Science
the University of Auckland, 2024.

School of Computer Science
The University of Auckland
New Zealand

An empirical study investigating the use of Python identifier names

Siqi Tong

April 2024

Supervisors:

A Prof. Ewan Tempero

Abstract

Understanding software programs is very difficult and time consuming. Identifier names have an important role in the source code, so it is an important part of enhancing comprehensibility. Many studies have demonstrated that using meaningful identifier names can improve the comprehensibility of programs. However, most of the naming conventions are rather general. We believe that more specific naming recommendations may be needed depending on the different cases. In this thesis, we investigated 745,651 identifier names from 100 open source Python projects from 6 different domains. We explored the connection between naming conventions and 9 naming practices and their differences in different contexts, such as loop statements, the size of scopes and different domains. Our results show that the use of identifier names does vary across cases. Making more detailed naming conventions based on different cases can help programmers to choose names that more accurately describe the concept of identifiers and thus improve comprehensibility.

Contents

1 Introduction

1.1 Program comprehension

The cost of software development and maintenance is very high. Reading and understanding the source code cost at least half of the time in the maintenance [48]. Comprehensibility is an important part of software maintenance. Identifier names make up 70% of the source code [16]. This means that the choices of identifier names can affect comprehensibility. There are proposed guidelines for how to choose good identifier names to help with comprehensibility. One way to determine if developers trust these guidelines is to determine to what degree developers follow them. The general goal of this thesis is to understand to what degree developers follow these guidelines.

Studies have been conducted on developer adherence to naming guidelines in a number of languages [5, 10, 24]. However there has been no study in Python. We believe this thesis can help fill the gap in this area.

Some studies have shown that choosing good identifier names would have a positive effect on comprehensibility and also can reduce the cost of software maintenance [18, 19, 34]. This is because that before modifying or adding new features, programmers must read and understand the meaning of the code in the projects related to the new contents [53]. As mentioned in the study by Deissenboeck et al. [16], although programmers agree with using good identifier names, choosing the most appropriate name is still complex. This is due to the lack of consensus among programmers. Even when facing to the same

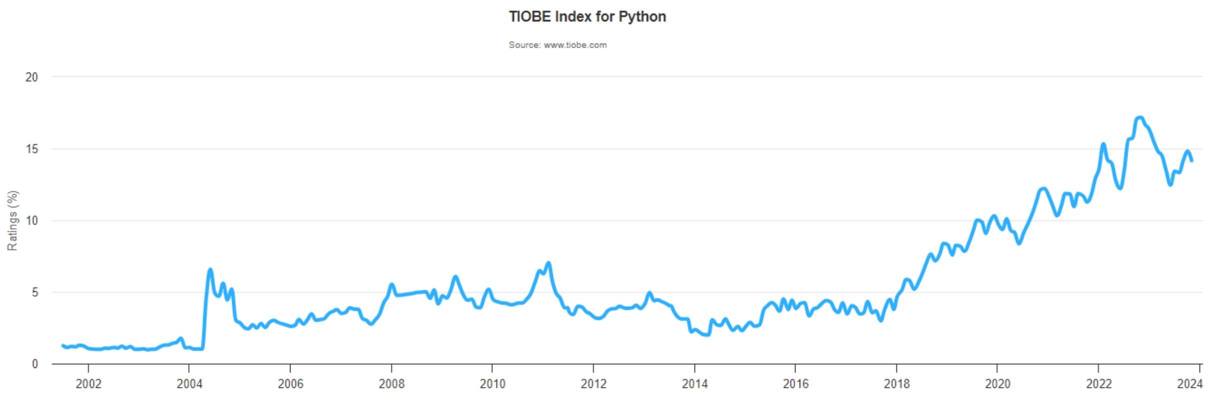


Figure 1.1: TIOBE Programming Community index: Evolution of the popularity of Python [58]

concepts, different programmers would use completely different words to describe it [20]. This causes confusion. In Clean Code [39], it is also mentioned that programmers should use meaningful identifier names to avoid confusion. This raises the question: `What is a meaningful name?` and `How to determine it?` Appropriate identifier names allow programmers to understand the high-level meaning of the concepts that described by the names [38]. The maintainability of code is closely related to the quality of identifier names, leading to a consensus in the field that meaningful names should be used [2].

Binkley et al.'s study [6] mentions that reading and understanding source code is fundamentally different from that in natural language. Although the grammatical structure of source code is very different from natural languages, programming languages are not entirely unnatural [37]. This is especially true for identifier names. A survey showed that 78.72% of the professional software developers involved strongly agree or agree that identifier names should follow the correct grammatical structure [2]. This might mean that using the identifier names that have correct grammatical structures might be easier to understand. There are some guidelines related to grammatical structures. For example, using verb phrases for functions/methods and using noun phrases for classes [39]. In addition to the guidelines related to grammatical structures of identifier names, there are many other different guidelines. For example, avoid using single letter identifier names and use dictionary words to name identifiers. Generally, comprehensibility is improved by programmers adhering to these guidelines when naming the identifier names. We believe that the quality of guidelines is correlated with comprehensibility.

1.2 Motivation

We found that there are many empirical studies on identifier names from different aspects. We think these studies usually have limitations.

The first limitation is that there are few studies related to Python among them. We found more studies related to Java [10, 11, 23, 34]. However, there are very few studies related to Python. This might be related to the nature of Python as a dynamic language. This means it is hard to get the types of identifiers in Python source code since the structure and types can still be changed during run time. For example, the types of variables and the output types of functions/methods. We think this might be the reason that there are very few studies related to Python in the field of program comprehension.

Names might be different in different programming languages. In Python code, it is possible that programmers choose identifier names that are different from those of other programming languages. We think it is still necessary to investigate type-related naming practices and also other aspects of the identifier names in Python repositories. For example, the length of identifier names, grammatical structures and also dictionary words. This is because Python is currently used in a variety of domains such as AI-related and web repositories. The Figure 1.1 shows the TIOBE programming community index which is used to measure the popularity of different programming languages. Python has become more popular in the last ten years, and it is also the most popular language in 2023. This makes us think that dynamic languages, such as Python, should not be ignored.

The second limitation is that the choice of identifier names could be very different in different situations. The names used as variables, functions and classes are all different [30]. However, most studies have usually focused on only one of these, such as function/method [2, 24] and variable names [5]. We believe that it is also necessary to compare these constructs to find out their actual differences in source code. In the empirical studies by Gresta et al. [23, 24], it is mentioned that the choices and uses of identifier names may be influenced by the context of the source code and properties of repositories. This thesis is inspired by their studies. We found that in two different For loops, programmers seem to choose different names. For example, in the For loop using `range`, it is more common to use the letter `i` as the loop counter. This does not exist in the For loop using `list`. Their study [24] investigated some specific naming practices used as variables, parameters and attributes in Java and C++ projects. We are going to discuss their studies in more detail in Chapter 2. We think it is a good idea to do a new empirical study that examines certain naming practices according to the characteristics of Python and also includes function/method names and class names to fill the gaps in the field.

In this thesis, we are interested in whether programmers follow certain guidelines in the open-source Python code and if they are related to the different contexts and repository properties. Our results show that most of the identifier names are consistent with the guidelines and the use of identifier names does vary across cases.

1.3 Thesis outline

This thesis is structured in the following chapters:

Chapter 2 presents and summarises the related works in comprehensibility and naming research area.

Chapter 3 describes the methodology of collecting source code, preprocessing data set, extracting identifier names, and categorising.

Chapter 4 shows and discusses the application of 9 naming practices in different cases.

Chapter 5 discusses the overall results to answer the research questions.

Chapter 6 is the conclusion and points out future directions.

2 Related work

There are two complementary types of research in the existing research on software comprehension which are empirical and technical [55]. Empirical studies investigate existing code in order to understand the programmer's intentions and the cognitive processes involved in programming the code. Technical studies are about developing tools that help programmers understand code or write code that is easier to understand. Theories and experiences drawn from empirical research are validated to provide suggestions to help develop more efficient tools. They form a feedback loop.

Through the two researches mentioned above, a large number of different theories and research methods have been developed over the last few decades. For example, Brook et al.'s [9] Top-down comprehension, Bottom-up comprehension [44, 52], and Letovsky's [36] combination of the both. Top-down comprehension refers to the step of mapping existing knowledge to source code. Bottom-up does the opposite in that programmers infer the semantics of the overall code by understanding individual lines or sections of code first.

This thesis is also relevant to the psychology and cognition of programmers since this is an empirical study. Past research has discussed that studies related to programmer cognition vary widely in their minutiae. Generally they contain four key elements: external representation, a knowledge base, a mental model and an assimilation process. These models describe the process by which programmers understand code [43] (Figure 2.1).

Firstly, external representation refers to all external information that is useful to the programmer, including but not limited to documentation and the source code itself. Sec-

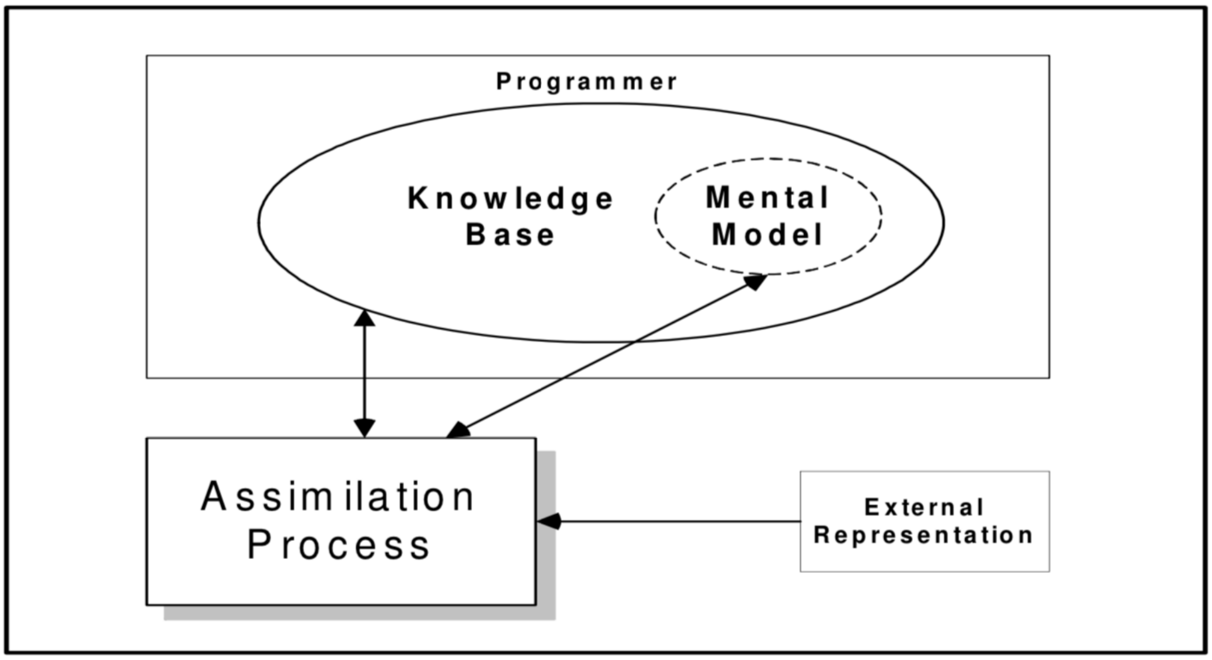


Figure 2.1: Program comprehension model (Michael P O'brien [43])

ondly, knowledge base refers to the skills and knowledge that the programmer possesses before understanding the code. For example, the understanding and consensus of a certain domain. Then, the mental model is a special type of knowledge base that refers to the current understanding of the program. It expands as the programmer's understanding of the code increases. The process of augmenting understanding and expanding the knowledge base with external information is known as the assimilation process [43]. The longer time it takes to read and understand identifier names will increase the time needed for the assimilation process. In our thesis, we investigate whether identifier names in source code are related to their associated contextual information. We believe this can help to understand the relationship between the concepts of identifier names and programmers' knowledge base.

There are relatively few studies on Python in the field of software comprehension. Vavrová et al. [59] analysed 32,058,823 lines of Python code and compared the results with those of Java by developing a tool to detect code smell. Their results show that Python is different from Java in this respect. For example, Python is relatively more prone to code smells with long methods, and less prone to too many parameters. We think this might mean that programmers have a different choice of identifier names in Python than in Java. Therefore, we think it is a good idea to do a a study of identifier names in Python code. It might be helpful to understand the naming conventions in Python.

In this thesis, we will mention 3 terms. They are naming conventions, naming suggestions and guidelines. In general, naming conventions mean that programmers think certain names are good, so they choose to do the same things. From there, they might become to be naming suggestions and then become to be guidelines. These 3 terms are inter-changeably in this thesis.

2.1 Proper names

As mentioned in Chapter 1, identifier names can be made up with standard English words. Deisenbock et al. [16] mentioned that the structure of identifier names is usually a compound of words on the left side, modifying words on the right side. These choices are more compatible with cognition [37]. In the programmer's society, it is widely accepted that some identifier names are better. Choosing proper identifier names is still challenging. Since identifier names do not affect the operation of the program, the use of any names is permitted. This includes but is not limited to full dictionary words, abbreviations, single letters, made-up words and combinations of multiple words.

In identifier name related research, it is widely mentioned that a proper name should be meaningful. As well, following naming conventions in projects improves maintainability [4, 12, 35, 42, 57]. Most programmers also believe that conventions need to be followed [2]. However, creating identifier names that adhere to the guidelines does not necessarily mean that they are meaningful. We believe that depending on the context and domain, more detailed and contextualised guidelines should be used. Parts of the information of identifiers might be included in the contextual information. For example, the use of a single letter `i` in a loop statement is acceptable [5].

There are two problems related to meaningful names. The first problem is that few researchers have explained what kind of name can be called meaningful. The second problem is that natural language is inherently ambiguous, such as the uses of synonyms, polysemy and also homonyms [20]. In the English language, which is the most commonly used in naming and also the subject of this study, this problem is completely unavoidable.

In Clean code, Martin and Ottinger refer to the need to use names that are clear and intended to reveal [39]. This would help to communicate to developers the purpose of the source code and its behaviour and eventually enable them to understand its true meaning [28]. Specifically, the following two points are made.

Firstly, avoid using words with double meanings whenever possible to avoid ambiguity about the meaning of the name when maintaining the code. Using double meaning words can lead to semantic defects, which is an error related to the meaning, semantics or logic of the program. A semantic defect usually occurs when a developer writes code that fails to properly understand the requirements of the problem or correctly implements the necessary logic. These types of defects may cause unexpected behaviour during program execution or failure to perform certain functions as expected. It may include incorrect use of variables, wrong conditions and also logic errors. This issue is also known as language anti-patterns (LAs) [3]. Their study suggests that unintended practices are likely to lead to misinterpretation of the source code and lead to wasted extra time. Deissenboeck and Pizka have constructed a source code framework based on bijective mapping of names and concepts to avoid this problem [16]. The definition of meaningful names and their impact on programs is still to be discussed. However, this problem is effectively mitigated by using meaningful names [28]. We are going to discuss this in Section 2.2.

Secondly, avoid using names with similar meanings and numbers, as this can also lead to confusion. For example, using a single letter as a name can be difficult to locate in the code, as it can appear anywhere in the code.

Finally, unsearchable and unpronounceable names can also be poor practices. For example, using a single letter as a name is difficult to locate in the code, as it can appear anywhere in the code. In most cases, programmers work in groups. The problem with unpronounceable names is that they are difficult to describe during the code review process [39].

In the study by Feitelson et al. [20], they conducted a series of multiple experiments related to the choice of names. They constructed 11 specific programming scenarios consisting of 47 naming questions in total. The experiment consisted of 334 programmers who could be professional programmers or computer science students. However, the median number of times that two programmers gave the same name was only 6.9%, which means that for the same instance, different programmers may give very different descriptions. This may be related to their knowledge base and personal experience. For example, people who are not native English speakers may be more likely to choose names that are not good enough.

It is important to note that choosing a different name is not the same as using a name that creates ambiguity. Different identifier names can refer to the same concept. This might be because different names can also used to describe the same concepts. In real programming environments, because of contextual information, even if other programmers use different but reasonable names with similar meanings, most of the time this will not affect comprehensibility. Another possible reason is that most people are unable to choose the most proper name for identifiers. There may be only one most accurate name but there are many wrong choices. A proper identifier name needs to make different programmers associate the same concepts when they read it. This is also known as consistency of identifier names [16, 51].

Therefore, a proper name should clearly describe the concepts it refers to [31]. It should also be able to associate the same concepts with the name when read by different programmers. The maintainability of code is closely related to the quality of identifier names [2]. The quality of names is not limited to the need to be meaningful, but also still takes into account other factors that may lead to reduced productivity. Therefore, choosing meaningful, searchable and pronounceable names can effectively reduce the time spent on maintenance. This can improve the efficiency and reduce costs.

2.2 Naming conventions

The research community has developed and recommended to programmers a wide range of naming conventions [2]. This would help programmers to use identifier names that can express the concepts more accurately. Pavlutin [17] explores the importance of some practical naming conventions for readability.

As mentioned in Section 1.2, Gresta et al. [23, 24] investigated identifier names in 50 Java and C++ repositories respectively by empirical studies. The study investigated 8 naming practices related to naming conventions in different context. It only investigated identifiers that are variables, attributes and parameters. The 8 naming practices are listed below: **Kings**: Identifier names ending with a number; **Median**: Identifier names that contain a number in the middle; **Ditto**: Identifier names with the same spelling as their types; **Diminutive**: Identifier names that are part of their type names; **Cognome**: Identifier names that contain a prefix or suffix of their types; **Shorten**: Single letter identifier names that are acronyms of their types; **Index**: Other single letter identifier names which use arbitrary letters; **Famed**: Most common Identifier names. These 8 naming practices can be classified into 4 types. They are number-related, length-related, type-related and common words.

For the first type of naming practices, the two types of naming practices Kings and Median are related to numbers used in identifier names. Number in middle and number at end are discussed separately because of their different uses [24].

An exploratory study on the use of numbers in identifier names of Java repositories was conducted by Peruma et al. [45]. Their study classified the meanings expressed by numbers in identifier names into 6 categories which are listed below: **Auto-generated**: Identifier names are automatically generated by code generation tools. The numbers are arbitrary and have no meanings. **Distinguisher**: Numbers are used to distinguish between two identical concepts in the same domain. It is usually used as a suffix to identifier names which is called number at end in this thesis. **Synonym**: Numbers are used to replace words that have the same pronunciation. The common used pairs are `2 and to`, and also `4 and for`. This function is used to shorten the length of identifier names. **Version number**: Use a combination of `V + number` to store version-related information in identifier names. **Specification and Domain/Technology**: These two kinds of numbers are contained in a specific concepts which can be a known specification and domain/technology names. For example, the word `2D` refers to 2-dimensional. In this thesis, we combine these two uses of numbers and call them special uses. The study concluded that most of the time, identifier names with numbers are also meaningful.

For the second type of naming practice, in the study by Gresta et al.[24], the only naming practice related to the length of identifier names is a single letter. Single letter names are acronyms which are the names of length 1. There is a lot of research related to the length of identifier names [50]. It includes the number of characters (length) in the identifier names, the number of words and dictionary words. Length-related topics are one of the most popular research topics in comprehensibility field.

There are many studies suggest that name length is related to whether the name clearly describes the concepts it refers to. Schankin et al. [2, 6, 40, 50] experimentally compared names of different lengths. They found that longer names are more descriptive and have a positive effect on comprehensibility. However, using method names that are considered to be too long is a code smell [21]. There are a lot of different opinions about the length of identifier names but there is no exact suggestion for the length of identifier names.

Although longer names are more descriptive, they would increase the reading time. There should be a fixed maximum length for identifier names. The length of identifier names is affected by the naming style they use and also the length of words that are not fixed. For example, names using under_score will be longer on average than names using camelCase. However, the difference in the effect on comprehensibility between the two is almost none [6]. The human short-term memory maximum is 7 $\pm$ 2 words [40]. Memorization and differentiation between similar names become more difficult when an identifier name has more words in it. In the survey of alsuhaibani et al. [2], 81% of the professional programmers involved think that method names should use 5 $\pm$ 2 words. This leads us to believe that investigating both the length and also the number of words used in identifier names is necessary.

There are many studies mentioning that the use of full words and dictionary words improves maintainability compared to abbreviations. Full dictionary words have exact definitions [16]. This means that using other words that are not defined can cause confusion and misunderstanding. Scanniello et al. [49] investigated in a controlled experiment whether the use of full words and abbreviations has an impact on the ability of novice software developers to identify and fix bugs in source code. Their results show that there is no significant difference between full words and abbreviations. Takang et al.'s study [56] showed that there is no statistical evidence on objective tests that using full words was easier to understand than abbreviations. However, participants felt that full words are easier to understand compared to abbreviations. Lawrie et al. [32] argue that identifiers must be named using full words.

The study by Hofmeister et al.[28] found that using full dictionary word names can increase the speed of understanding code during maintenance by an average of 19%. As we mentioned above, single letter names are special abbreviations (acronyms). They are sometimes discussed separately from abbreviations in research.

Lawrie et al. [34] designed a controlled experiment with three groups. Each of the three groups of programmers is required to read and understand 12 functionally identical functions but with names that use full words, abbreviations, and acronyms. However, we believe that there are limitations to this study. The variables, attributes and parameters in any arbitrary functions/methods are usually not all named in single letters. It is not usual to name different identifiers with the same number of letters at the same time. This means that it is unlikely there would be confusion between two different single letter identifiers. IIn this case, it is impossible to see the names but not remember what the concepts they refer to are. This is because that the functions used in this study are small and the location of their first assignment can be found quickly. However, in a real programming project, it is more difficult to find the locations of declarations in large scopes.

Beniamini et al. [5] investigated variable names in 1,000 popular Github repositories of 5 programming languages. Programmers generally think long descriptive names are meaningful. However there are some other letters besides the commonly used `i` in loop statements that are widely used. The use of these names might be consistent with another meaningful explanation mentioned in Section 2.1 that programmers would think of the same concepts when they are reading the single letters. The study also suggests that single letters can be used in specific contexts to make the code more concise.

For the third type of naming practices, the results of the survey of type-related naming practices show that programmers would save type-related information in identifier names in both Java and C++ repositories [23, 24]. This is because using type-related names can prevent mental mapping which is typically connected to the concepts of the problem domain. The use of acronyms and abbreviated type names in names reduces the overload of reading. This also helps programmers to remember the type of the identifiers, especially when using acronyms [22]. Moreover, single letters related to type are not necessarily abbreviations of type. For example, the use of letters `i, j, k and n` are more likely related to integers, and the use of letters `d, e, f, r, and t` are related to floating points [5, 10].

As mentioned in Chapter 1, in contrast to Java and C++, Python is a dynamic language that the types do not have to be fixed. We need to investigate whether these practices also occur in dynamic programming languages. As an alternative, in this thesis, we investigated words that are commonly used in identifier names which is called most used words. This will be discussed in detail in Section 3.2.

The last type of naming practice mentioned in Gresta et al.'s study [24] is about common words. This might be because the common words exist in the programmers' mindsets and can be easily remembered and comprehended. The problem is that using too many same names in different ways in the same repository can be confusing and difficult to search. However, common words are more likely to point to the same concepts

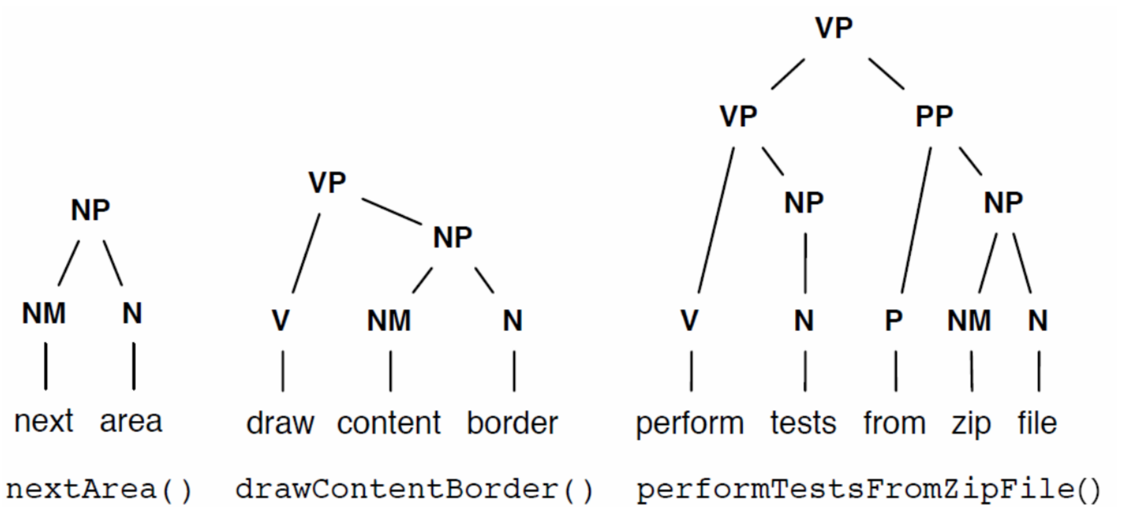


Figure 2.2: An example of noun, verb and prepositional verb phrases [41]

in different programmers' mindset [39]. For example, when they see the letter `i` in a loop body, they would associate the letter to a loop index. We believe that not using them in the same scope would not have negative impacts on comprehensibility.

In addition to the 4 types of naming practices mentioned above, grammar-related ones are also widely mentioned in the relevant literature [26, 27, 37]. Newman et al. [41] mentioned that different types of identifier names (variables, parameters, attributes, functions/methods and classes) have different grammatical structures. Figure 2.2 shows some examples of possible structures. Among them, we note two naming conventions related to grammatical structures. They are the use of verb phrases for naming functions/methods and the use of noun phrases for naming classes [39].

Caprile and Tonella [13] analysed the grammar of function/method names and found several patterns. Most of them exist with verb phrases but not all names do. We think it is necessary to investigate the reasons why they are not verb phrases and to judge whether these usages are reasonable. Liblit et al. [37] considered that functions/methods to be used to compute or perform an action, so verb phrases should be used. More specifically, Abebe et al. [1] suggested that function/method names should start with a verb. They argue that this approach can emphasis the action involved in functions/methods. Grammar-related contents are not mentioned in the style guide for Python code (PEP8) [25], but this does not mean that programmers are not following the suggestions from other guidelines.

Gupta et al. [26] developed a part-of-speech tagger (Posse) to analysis the grammatical structure of identifier names in source code. Through their analysis, they found that using noun phrases is typical for attributes and also classes. Singer and Kirkham [54] suggested that the grammatical structure of classes should be `(objective)* (noun)+` which is a noun structure. Butler's research [12] also found that 120000 unique class names in 60 Java repositories are mostly noun structures. The class names, that are not noun structures, still contain nouns. This might be related to the interfaces and inheritance to which they belong. Their studies focused more on the minor differences between different noun structures. This thesis focuses more on whether class names in Python are noun structures, and also the reasons for doing so or not.

As mentioned in Section 2.1, we believe that specific guidelines should meet the actual needs in different situations. In Gresta et al.'s study [24] of two object-oriented languages, the differences in choices of identifier names for different situations are discussed. It is categorised into two types. The first is the contextual information related to identifier names. The second is the external information related to repository properties.

For the first part, they only discussed the identifier names of variables, attributes and parameters. On top of that, they investigated the differences between the names used in While, For, If and Switch [24]. As mentioned in Chapter 1, we think this study should be extended to variables, parameters, attributes, functions/methods, and classes in Python. And also to investigate the different application of naming conventions between function names and method names.

Secondly, Gresta et al. [24] investigated identifier names in different Github repositories, which contain three repository properties which are lines of code (LoC), number of commits, and number of commiters. Their results show that in Java only Ditto, which uses the same spelling as type, is related to LoC. Smaller repositories are more likely to use Ditto. Whereas in C++, larger repositories use more names ending in numbers, smaller repositories are more likely to use single letters. In both languages, neither the number of commits nor the number of contributors has any relation to naming practices. In addition to these three, the year of the repositories may also influence the choices of identifier names. Lawrie et al. [33] mentioned that the quality of identifier names has improved over the 30 years from the 1970s to the 2000s. With the development of technology, the choices of identifier names are no longer limited by the size of the memory, especially for the length of the names [8]. This might also have an impact on naming practices. Therefore, we think it is necessary to add first release dates and number of releases in this thesis.

We believe that the categories of repositories also would have an impact on the choice of identifier names. This is not mentioned in the study by Gresta et al. [24]. For example, the single letter identifier names `x` and `y` would be more common in computer-vision repositories, which use coordinate axes. In our study, we categorised the 100 repositories into 6 categories related to common uses of Python. They are AI, web, scientific-computing, visualisation, computer-vision and automated tools which will be discussed in detail in Chapter 3.

3 Methodology

In this chapter, we are going to discuss how the data is collected and analysed in this thesis. We investigate how python programmers name identifier names in repositories through empirical research. Empirical research is a form of academic study that uses direct and indirect observation or experience to gather evidence and deepen our understanding of the world. This thesis aims to investigate programmers' compliance with selected naming advice in a variety of contexts across a wide range of Python repositories. In this study, we use quantitative research methods to statistically analyse the data collected and to answer the empirical questions.

Specifically, we collected a total of 745,651 identifier names from 100 Python repositories. We also investigated the 9 naming practices that are related to naming conventions (Section 3.2). In this thesis, we are going to discuss and present the data from multiple perspectives in order to address and answer research questions mentioned in Section 3.1. We are going to introduce the methodology in the following sections. Section 3.1 describes the research question. Section 3.2 describes the 9 naming categories discussed in the thesis. Section 3.3 describes the process of collecting the data. Section 3.4 is about how the data is analysed.

3.1 Research question

We summarised what we discussed in Chapter 2 and formed 3 research questions.

3.1.1 RQ1: How are the different naming conventions used in practice?

We investigated the use of the 9 naming practices related to naming conventions in 100 Python open source repositories from Github.

3.1.2 RQ2: How much influence does context have on the choice of identifier names?

We investigated the 6 different kinds of identifier names, which are variables, parameters, attributes, functions, methods and classes. There are also sub-categories related to context. For example, loop statements (For, While), If statements, the scopes and scope length. We investigated if there is any relationship in name selection in different context.

3.1.3 RQ3: Is there a difference in the choice of names in different Python repositories?

We investigated if repository properties would affect the choice of identifier names. They are lines of code (LoC), number of commits, number of contributors, number of releases and first release dates.

We classified the repositories into 6 categories according to the relevant domains. They are AI, web, scientific-computing, visualisation, computer-vision and automated tools.

3.2 Naming conventions and naming practices

Python is a dynamically typed language. It is characterised by checking the type of data in the code only at run-time and determining the type of a variable or object based on the type of its value. This differs from statically typed languages such as Java and C++, which check data types at compile time. In this thesis, we examined the 9 representative naming practices for this characteristic of Python. In the following part, we are going to list and describe the 9 naming practices. We will also explain the reasons for the choices and what might make programmers choose to practice these in their repositories.

Single letter: This category refers to naming identifier names with a single letter which includes both uppercase and lowercase letters. In the process of actual programming, single letter names are difficult to search. Using single letter names in large scopes, such as global variable, is not good for comprehensibility of the code [5, 24].

Single letter names may be appropriate for small scopes. In small scopes, programmers are less likely to forget the concepts to which the letter refers. This is because where the identifier is used is close enough to where it was created. Programmers can more easily find and understand the reasons for using the letters. Programmers are also used to use letters `i`, `j` and `k` as loop counters even when the loop body is large. This is the consensus that exists in the knowledge base. These letters are also commonly used as loop counters in both reference books and introductory textbooks. For example, `Python phrasebook: essential code and commands` [15] is a reference book and `Python3 object oriented programming` [46] is an introductory book which introduced the use of single letter `i` as loop counters.

It is also used in Mathematics-related fields. For example, `x`, `y`, and `z` are letters commonly used for this purpose. Programmers involved in a project would have some knowledge of the field. They do not need to spend extra time to understand the meaning of these letters. This is another consensus that exists in the knowledge base.

The problem of single letter names is that it is still possible for a single letter to refer to multiple concepts in a given field. For example, in Web repositories, `f` may refer to files or forms. The use of a single letter in such cases would cause confusion.

Number at end: This category refers to the use of numbers at the end of identifier names, such as using `num1` and `num2` as parameters and using `list1` and `list2` as list names.

The suggestion from Clean Code [39] is to use names with numbers only when necessary. This means we need to discuss the different cases of using numbers in identifier names. As mentioned in Section 2.2, in the study by Peruma et al. [45], most of the types of names mentioned in relation to numbers are associated with numerical endings except synonyms. They are arbitrary numbers (Auto-generated numbers), distinguishers, versions and specific uses.

For the first category, we believe that using arbitrary numbers as endings is a very poor choice. This is because it is useless for describing the concepts of identifiers and also increases the length of identifier names. It also causes the programmer reading the code to be confused about what the string of numbers means, thus slowing down the reading speed and leading to less comprehensibility.

For the second category, we believe that distinguishers are meaningful. These names usually end with a single digit, as in the examples we mentioned earlier, `num1` and `num2`. When we use these names, we usually do not care about their specific identities, but rather about the concepts they refer to in the prefix name. For example, parameters are used in a function that compares the size of two numbers. It would be difficult to find a better choice in this case. We have considered the 2 pairs of names `num_one`/`num_two` and `num_a`/`num_b`. The 2 pairs of names do not contain numbers. We do not think this contributes to comprehensibility, but rather makes the names longer and harder to read. We think using numbers in this case is visually clearer.

For the third category, in Chapter 2, we mentioned that version-related endings increase the length of the names but do not help with understanding the code. This information is usually not related to the concepts of identifier names. Therefore, we think it would be better to keep the information about the version as a comment if there is a need for it.

For the fourth category, we believe that numbers that are considered to have a specific purpose are meaningful. For example, `web3` refers to the next iteration of the World Wide Web and `95` refers to the confidence interval. For these names, there are no better alternative names could be found. Even converting `95` to the full word `ninety_five` would not contribute positively to comprehensibility. Using numbers are more obvious in these cases.

Number in middle: This category refers to the use of numbers in the middle of identifier names such as float64_matrix and x1_shape.

All the types of names related to numbers mentioned in Section 2.2 can be related to number in middle. These types are arbitrary numbers (Auto-generated numbers), synonyms, distinguishers, versions and specific uses.

The synonyms are used to shorten identifier names and make connectives more intuitive. However, they are unpronounceable and can not be distinguished from words not replaced by synonyms when communicating with other programmers.

The distinguishers, that are number in middle, are different from number at end. This is because the part it uses to differentiate is usually a phrase containing 1 or more numbers. For exmaple, the two method names `nhwc3to4` and `nhwc3to8` in`AITemplate`. The reason for using longer phrases is related to more complex concepts. This means that there might be identifiers, that need to be distinguished, are not close to each other in the source code. For example, the two methods names mentioned above are more difficult to detect as corresponding to each other than the common parameters in number at end.

The similar identifier names would make it difficult to search in the code. The suggestion is to avoid using names with numbers extensively and to ensure that the other parts of the identifier names consist of meaningful words [39].

Dictionary word: Dictionary word is a word that can be found in a standard dictionary. As mentioned in Chapter 2, the guidelines state that identifier names should be named by dictionary words. Past studies have pointed out that using full word identifier names can effectively help programmers understand the meaning of the name and relate the context [23, 24]. This can effectively reduce the time to evaluate and maintain the code. However, in the actual code, there are numerous abbreviations are still tacitly accepted, such as URL for Uniform Resource Locator, and db for database. These abbreviations are often well known.

For this naming practices, we are interested in the possible reasons for using these identifier names with non-dictionary words and whether there is an impact on comprehensibility. We explored 4 different scenarios which are listed below: **All**: all the words in the name are dictionary words **At least one**: at least 1 of the words in the name is a dictionary word **All but one not**: Only 1 of the words in the name is not a dictionary word **None**: All words in the name are not dictionary words

Most used names: This category contains the most commonly used names in the programs. These names usually appear in some default situations, such as using result to refer to the return value of function. However, using multiple identical names could lead to reduced comprehensibility and also cause difficulties in searching and locating the name in the code. Although these names are usually in programmers' knowledge base, there may also be names that more accurately describe their concepts [23, 24].

Most used words: This category contains the most commonly used words in identifier names. We investigated through this naming practice whether programmers also use words related to identifier types to name identifiers in a dynamic programming language which is Python in this thesis. The benefits of investigating this naming practice will be discussed in Chapter 5.

Verb Phrase for function/method: This suggests that method names should contain a verb or verb phrases since a method should contain an action [2, 39]. When programmers read an identifier name that contains a verb structure, they can intuitively realises that it is a function/method and determines its functionality.

But there are actually exceptions in practice. For example, a class has an attribute named `size` and a method also named `size()`. In this case both `object.size` and `object.size()` would return the same value. Even if the method is a noun phrase, its functionality, that is getting the size of the object, is relatively obvious compared to other cases. In this thesis, we investigated identifier names that do not contain a verb structure and also analyse possible reasons. We are also interested in the reasons for using verb phrases in the names of other kinds of identifiers and if these practices would cause comprehensibility problems.

Noun Phrase for class: This suggests that class names should be a noun or a noun phrase, since a method is a verb that occurs on the object represented by the class [39].

In addition, variables, parameters and attributes do not contain actions, so they are usually also noun phrases. We also investigated how they differ from the noun phrases in classes.

Length and number of words: As mentioned in Section 2.2, The length of the identifier name is a widely discussed and still controversial topic. In terms of maintainability, using excessively long names is a form of code smell [21]. Although there seems no straight relationship between the complexity of the concepts of identifiers and the length of identifier names, accurately describing a conceptually complex identifiers might require using longer names. We think it is a good idea to investigate the length of names that

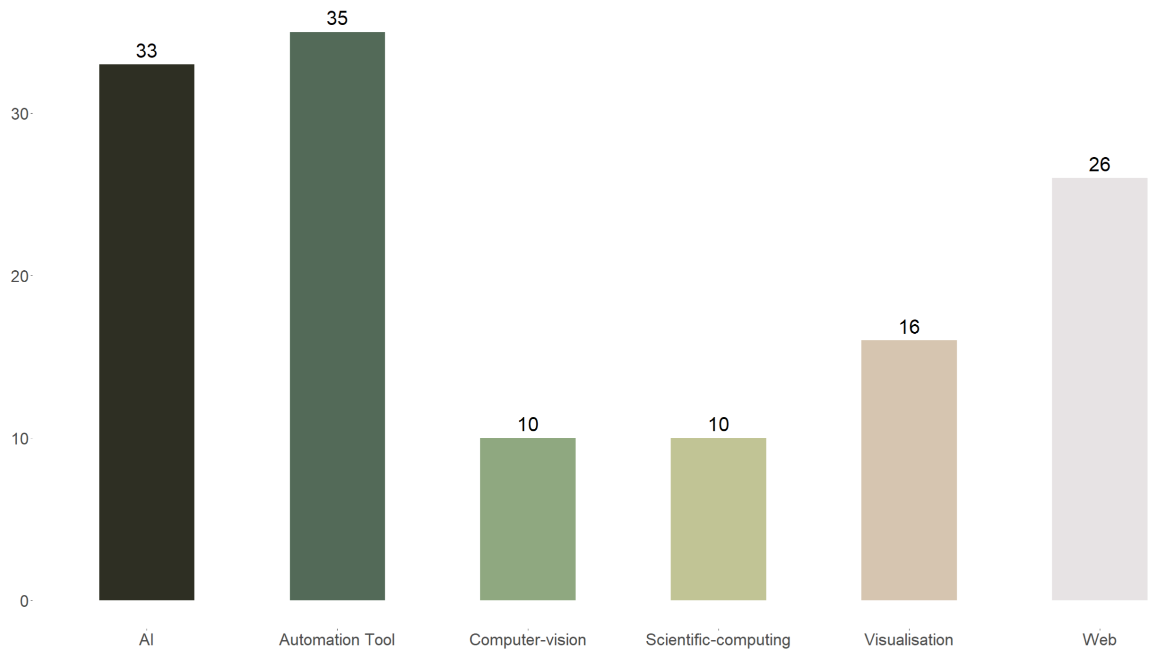


Figure 3.1: The categories of the 100 Python repositories used in this thesis

programmers need to describe different concepts in the actual source code.

Because of using different naming styles would influence the length of identifier names, we investigate both the length and also the number of words used in the identifier names in this thesis.

3.3 Data Collection

3.3.1 Repository selection

The data for this study came from 100 Python repositories on GitHub. These repositories are all based on English as the language used for programmers to make naming decisions. English is the most widely used language in code [10]. Using English-based repositories in our study also helped increase comparability between the data since it becomes difficult to compare similarities and correlations between identifier names in different natural languages. For some of the naming practices mentioned in Section 3.2, we use natural language processing (NLP) to do the analyse. These repositories cover a number of different fields. We have categorised these into 6 different categories. They are AI, web, scientific-computing, visualisation, computer-vision and automated tools (Figure 3.1). A repository may belong to more than one repository category since they can be cross-disciplinary. Among the 100 repositories, 30 of them belong to multiple categories. We made this decision because we want the repository categories to have adequate data and we also want to choose repositories from different categories.

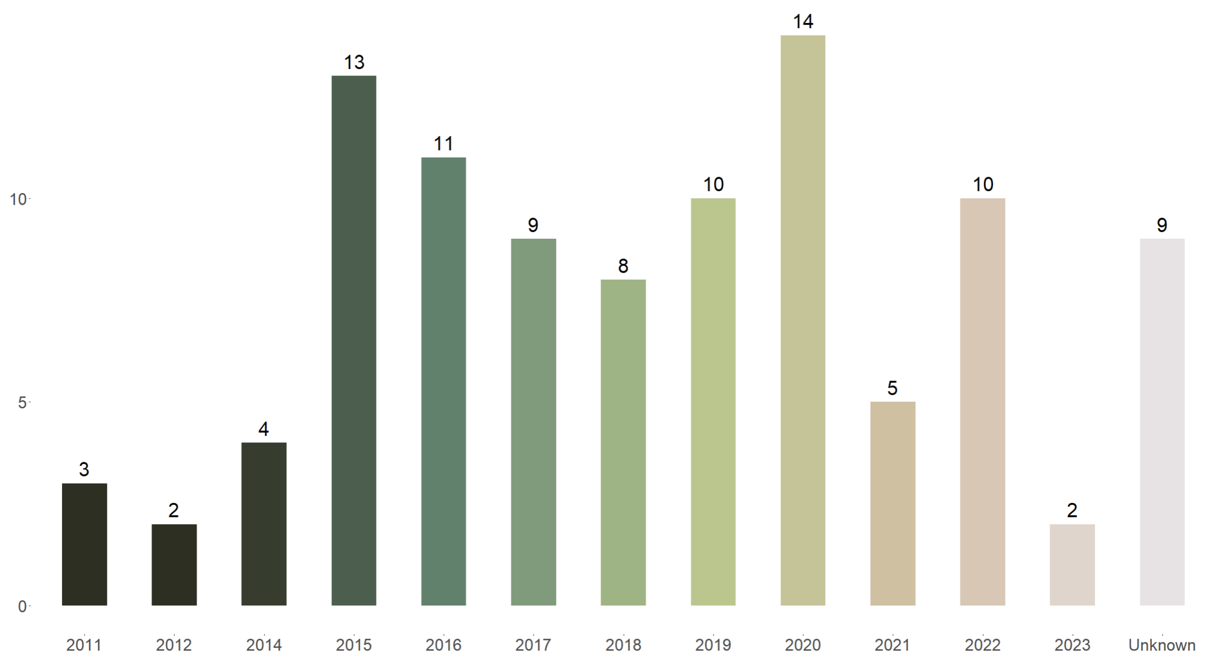


Figure 3.2: The distribution of first release year of the 100 repositories

We have included repositories that are popular in a variety of fields, all of which have over 2,000 Github stars. We believe that repositories with this amount of stars are representative enough and widely used in their field. We did not include forked repositories in our selection. Our selection of repositories also includes both small repositories less than 500 lines of code (LoC) and large repositories greater than 100k LoC. We also consider the diversity in the number of commits and the number of contributors in the selection of repositories.

The year in which the repositories were created is also taken into account. In this thesis, we use the `first release date` on Github as the year in which the repositories were created. We investigated the number of releases of these repositories. If the repository does not use this feature, the date of the first stable release in `tags` is used and also the number of tags are used. Out of 100 repositories, 9 do not use `releases` or `tags`. These repositories are excluded from the statistical tests about the year and number of releases in the following chapters. Programmers can upload unfinished repositories to Github, but the versions that are put into releases or tags are usually working beta or stable versions. This is the reason for choosing `releases` as the time that the repositories were created. Figure 3.2 has shown the first release year of the repositories used in this thesis.

We collected these repositories in a variety of ways. Firstly, we used the Github data mining tool developed by Ozren et al [14]. This tool provides over 12 different filters of four types: History and Activity, Popularity Filters, Size of codebase and Date-based Filters. Secondly, we include some of the packages used to develop our tools. We also investigated a number of recommended lists to find suitable repositories. As these

Python repositories are from different fields and have different features, we believe they are relatively representative.

The data used in this thesis are time-sensitive since the Github repositories might be updated through time. All the repositories and data used in this thesis were collected in October 2023. Of the 100 repositories, 77 have been continuously updated in the last 6 months. Five of them have not been updated in the last 3 years.

3.3.2 Data preprocessing

We preprocessed the collected repositories before extracting identifier names and its related information from the code.

Firstly, the name of .zip file from Github usually contains extra branch information, such as `numpy-main` and `manim-master`. This is because after October 2022, the original branch of any Github repository would use `main` as the default branch name. These are not the only options since some programmers might choose to use other suffixes, such as `dash-dev`. These endings are not useful for this thesis. Therefore, we removed them for all mentioned repository names in this research.

Secondly, We noticed that there are many irrelevant files in the repositories. The focus of this research is on identifier names used in Python, but some repositories may contain a small amount of code written in other languages. Therefore, only Python files ending in `.py` or `.pyi` were retained in preprocessing stage. In this step, we also removed all the document related files. These files hold important information about the use and maintenance of software development, but are not useful to this thesis. These folders and files are usually named `docs` or `document`, so we can detect the identifier names containing these related words and remove them.

There is a certain type of Python (`.py` or `.pyi`) file that affects the data in this study which is the code used for software testing. Testing is an essential process in software development. It is an important step used to test that parts of the code, and the software itself are as expected. However, this code may not be subject to the usual naming convention and will differ from the usual Python code in a number of ways. For example, professional programmers in the questionnaire mentioned that the function names used in tests tend to be very long because they need to be longer to explain what the test does [2]. Since multiple programmers may be responsible for quality assurance of a repository, some teams would upload and keep these test files on Github. They are not the actual code used in the repository, so we removed all folders and files containing the keyword `test` and `tst`.

3.4 Extracting identifier names in Python

3.4.1 Extracting names and related information

In this section, we will describe how to extract identifier names and their associated information from source code. Including the identifier names, we obtained 34 kinds of related information for each identifier which will be discussed in the following paragraphs. This information can be divided into four categories, namely the kinds of identifier names (variable, attribute, parameter, function, method or class), the scope of the names (repo, module and scope), the context of the names and the information related to the 9 naming practices mentioned in Section 3.2.

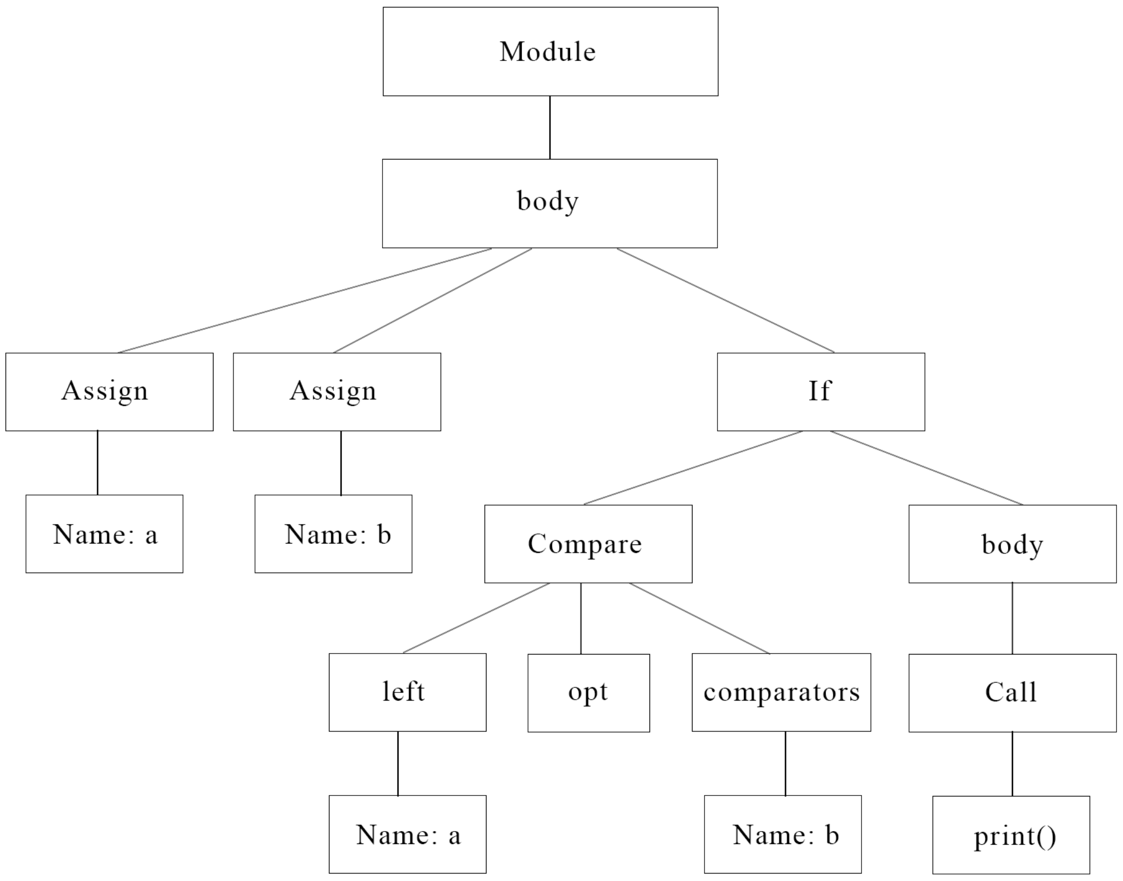


Figure 3.3: Abstract Syntax Tree Example Graph

```python
a = 10
b = 20
if a > b:
    print("a is greater than b.")
```

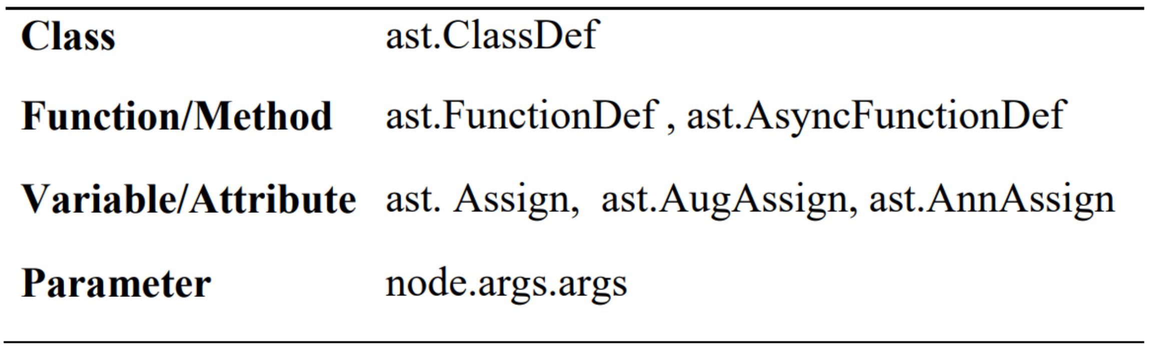
Listing 3.1: Abstract Syntax Tree Example Code

```
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store())],
            value=Constant(value=10)),
        Assign(
            targets=[
                Name(id='b', ctx=Store())],
            value=Constant(value=20)),
        If(
            test=Compare(
                left=Name(id='a', ctx=Load()),
                ops=[
                    Gt()],
                comparators=[
                    Name(id='b', ctx=Load())]),
            body=[
                Expr(
                    value=Call(
                        func=Name(id='print', ctx=Load()),
                        args=[
                            Constant(value='a is greater than b.')],
                        keywords=[]))],
            orelse=[])],
    type_ignores=[])
```

Listing 3.2: Abstract Syntax Tree Example

Firstly, there are 6 different kinds of identifiers are being discussed in this thesis. They are variables, parameters, attributes, functions, methods and classes. In order to get the identifier names of the 6 different kinds correctly, we used the Python Abstract Syntax Tree package [47] as the base to build a tool. AST is a representation of the abstract syntax structure of source code. It also includes the data used in this thesis that are identifier names and related information. For example, the code in Listing 3.2 will be divided into multiple levels in the AST output (Listing 3.1). All the identifier information is included in the tree structure (Figure 3.3), and they follow specific rules. Therefore, we can get the identifier information through automated tools based on the Breadth First Search algorithm (BFS).

BFS is originally used to search nodes in a tree structure. Since it can traverse all the nodes in a tree layer by layer, it can also be used in our tool. For example, `Assign` and `If` are 2 different nodes (Listing 3.2). Our algorithm adds the child of the non-variable node and its scope information obtained during traversal to a queue, since variable must



Class	ast.ClassDef
Function/Method	ast.FunctionDef , ast.AsyncFunctionDef
Variable/Attribute	ast. Assign, ast.AugAssign, ast.AnnAssign
Parameter	node.args.args

Figure 3.4: Identifier type - AST node

be a leaf in this tree structure. Scope information includes which of the module, class or function/method the node belongs to, scope name and the size of the scope. In this case, when traversing to the child node, the scope information to which it belongs can be stored in the queue by saving in a Python list. The scope name is used to avoid recording the same identifier repeatedly, which will be discussed in detail in section 3.4.2. The size of scope is the contextual information used to analyse whether the naming practice is related to it.

The 6 different kinds of identifiers (variable, parameter, attribute, function, method and class) have different types of nodes in the AST structure (Figure 3.4). We need to use a different way to get these identifier names. The same identifiers would only be recorded once, so our tool would ignore the most special one `ast.AugAssign`. AugAssign is the abbreviation of augmented assignment such as the code `x += 1`. Although the names of the variables would also appear in this case, this assignment will only work correctly if x is defined. We assume all the source code in the repositories used in this thesis has been tested and with no compilation problems. Therefore, the identifiers related to node ast.AugAssign should have been recorded before traversing this node, which is why the tool can directly skip it.

```
variable1 , variable2 = 1, 2
```

Listing 3.3: Multiple Assign

In python syntax, two variables can be assigned values at the same time (Listing 3.3). In this case the targets of `ast.Assign` node is different. Normally, the instance of targets is `ast.Name`, which means that `targets.id`, which is the identifier name, can be extracted directly. However, when assigning values to multiple variables in one line, targets is a list containing multiple ast.Name objects.

```
1  >>> imoport ast
2  >>> print(ast.dump(ast.parse('variable1 = 1; object.attribute1 = 1')
       , indent = 4))
3  Module(
4      body=[
5          Assign(
6              targets=[
7                  Name(id='variable1', ctx=Store())],
8              value=Constant(value=1)),
9          Assign(
10             targets=[
11                 Attribute(
12                     value=Name(id='object', ctx=Load()),
13                     attr='attribute1',
14                     ctx=Store())],
15             value=Constant(value=1))],
16     type_ignores=[])
```

Listing 3.4: Attribute - AST Output

There is another case relates to variables. Class attribute is a kind of special variable, and its expression in the AST structure is also different from ordinary variables. As shown on Listing 3.4, the identifiers of ordinary variable assignments will be expressed in the form Line 5 to Line 8, that is, `Name` and `id`. The assignment of attribute is expressed in the form of Line 9 to Line 15. Firstly, it has an extra attribute level in targets. Our tool would use this information to determine whether it is an attribute. Secondly, the `id` of attribute refers to the name of the object rather than its own name. This is different from the `id` of variable assignments as shown in Line 5 on Listing 3.2. Therefore, if it is determined to be an attribute assignment, then our tool would get the value of `attr` in Line 13 as identifier name which is `attribute1`.

In this thesis, we investigate 3 types of context-related information. They are loop (`For, While`) statements, `If` statements and scope length of functions/methods. Unlike other programming languages (e.g., Java or C++), there was no `switch case` feature in Python before version 3.10. Although the structural pattern matching was added after 3.10, switch will not be discussed in this thesis because not all the source code samples used in this study use versions later than 3.10.

```python
# range
for i in range(1, 4):
    pass

# list
list1 = [1, 2, 3]
for num in list1:
    pass
```

Listing 3.5: For Loop

As shown in Listing 3.5, there are 2 types of For loop statements can be used in Python source code. The first one uses range as the conditional statement. The second one uses a list as the conditional statement. In this thesis, we are going to call them `range type For loop statement` and `list type For loop statement`. As mentioned in Chapter 1, programmers might have different preferences for name selection when using these 2 types of for statements, so we discuss them separately.

The function/method refers to that the nodes of function and method in the AST are both ast.FunctionDef or ast.AsyncFunctionDef. However, our tool can determine whether it is a function or a method through the scope information recorded before. If the scope is module or function, it is a function. If it belongs to a class, it is a class method.

For the 9 naming practices mentioned in section 3.2, our tool can determine if the naming conventions are followed by the identifier names.

The first is the use of numbers in the identifier names, where we use a Boolean value to record whether the name is Number at the end and Number in the middle.

Secondly, there are naming practices related to the length of the name, which includes the length (the number of characters in the names), the number of words used, and information related to the single letter. For example, the length of the name `get_list_display` is 16 and the number of words of the name is 3. We not only record whether the name is a Single letter name, but also record whether the name is uppercase or lowercase and which letter is used for more detailed analysis.

In order to analyse the words used in identifier name and the grammatical structure. We use the spaCy package[29] to split identifier names that contain multiple words. This is a library can be used for advanced natural language processing in Python. For the dictionary word detection part, we use the PyEnchant package which is a spellchecking library for Python. It can be used to determine if the identifier name is a dictionary word by checking if it matches the spelling of the words contained in the English spelling library which can be treated as the dictionary. We record results not only for each individual word, but also for the entire identifier name in three cases. These are whether all words in a name use a dictionary word, whether there is more than or equal to one non-dictionary word, whether there is less than or equal to one dictionary word, and whether there is more than or equal to one non-dictionary word. It is more detailed to discuss these cases in categories for analysing names.

```
verb_phrase_pattern = [{'POS': 'VERB', 'OP': '?'},
                       {'POS': 'ADV', 'OP': '*'},
                       {'POS': 'VERB', 'OP': '+'}]
```

Listing 3.6: Verb Structure - Regular Expression

For part-of-speech detection we used `textacy` package which is based on the spaCy library. However, we need to preprocess the words since the training data used in `textacy` is from natural language in standard prose which is different and has more contextual information than identifier names. This problem would affect the correctness of the grammatical structure detection for function and class names in this thesis because there are some words that can be different part of speech in different contexts. For example, the word `register` can be both noun and verb, and both of them may appear as identifier names. Therefore, we take the preprocessing method mentioned by Binkley et al. [7] to add prefixes to words to make `textacy` can identify part of speech more accurately. Their empirical study has shown that names processed by this method have an 88% accuracy rate in identifying part of speech. The 2 naming practices related to part of speech in this thesis are the use of verb structure for function/method naming and the use of noun for class naming. For function/method names, we added the word `Please,` before the verb to make sure it is correctly classified as a verb. Since grammatical structure can be satisfied in different ways, regular expressions is used for `SpaCy` to distinguish verb structure (Listing 3.6). For detecting Noun phrases, we use the method `noun_chunks` from the `textacy` package.

There are some special identifiers in Python. For example, the method `__init__` is a constructor used to initialise a new Class object. The parameters `self` and `cls` are used to access the instance methods/attributes and classes respectively. They are not reserved words in Python, but we think that the use of these words is usually independent of the programmer's choices. Therefore, we removed these words before analysing our data set.

3.4.2 Dynamic variable name assignment

The Identifiers in the code have multiple properties. We are going to discuss 2 of them which are declaration line numbers and names in this section. As a dynamic programming language, Python is different from a static language. It does not require type declarations when assigning values to variables. In Python abstract syntax trees (AST), all variable assignments use ast.Assign. There is no declaration node as other programming languages such as Java, so there is no easy way to distinguish the first use of the identifiers in

Python. The declaration line number cannot be extracted directly. We need to process the identifier names extracted by AST to avoid recording duplicate names.

```
variable1 = 1
variable1 = 10
```

Listing 3.7: Variable Assignment

```
>>> import ast
>>> print(ast.dump(ast.parse('variable1 = 1; variable1 = 10'),
    indent=4))
Interactive(
    body=[
        Assign(
            targets=[
                Name(id='variable1', ctx=Store())],
            value=Constant(value=1)),
        Assign(
            targets=[
                Name(id='variable1', ctx=Store())],
            value=Constant(value=10))])
```

Listing 3.8: Variable Assignment - AST Output

In Listing 3.7, since `variable1` is assigned twice, 2 `ast.Assign` will be generated in the AST and their `id` are also the same. which is shown in Listing 3.8. This kinds of repeated names only show that the variable is used multiple times in certain scope (e.g. functions and classes). This has nothing to do with the choice of identifier names. We have added code to determine whether identifiers are repeated in the tool to prevent repeated recording of the same identifier. We will introduce how we choose the identifier names in several different cases.

```
variable1 = 1
def new_variable():
    variable1 =  1
```

Listing 3.9: Identifier Names in Different Scopes

Firstly, duplicate identifier names in the same scope are only recorded once, which is the case in Listing 3.7. The scope mentioned here includes module, function and class. For example, in Listing 3.9, the scope of variable1 in Line 1 is the module of the code and the scope of variable1 in Line 3 is the function `new_variable`. Even though both names are `variable1`, they are actually 2 different identifiers. As mentioned in 3.4.1, we will record the names of the scopes of all identifiers. Then every time `ast.Assign` is detected, the tool can automatically determine whether the identifier already exists in the scope through the scope names in the record.

```
1 variable1 = 1
2 def add():
3     global variable1
4     variable1 =  variable1 + 1
```

Listing 3.10: keyword global

We have considered a special case, which is the use of the Python keyword `global`. This keyword allows to reassign the variable outside the current scope. As shown in Listing 3.10, the scope of variable1 in Line 1 is the module of the code and the scope of variable1 in Line 4 is the function `add`. In this case, even if the two `variable1` are in different scopes, they are still the same identifier.

```
1  class Class1:
2      field1 = 1
3      field2 = 2
4      field3 = 3
5      field4 = None
6  
7      def __init__(self):
8          self.field3 = 3
9  
10     def method(self):
11         self.field4 = 4
12         self.field5 = 5
13 
14 object = Class1()
15 object.field6 = 6
16 object.field5 = object.field5 + 1
17 object.field6 = object.field6 + 1
```

Listing 3.11: Class Attribute

Secondly, in Python, class attributes are also dynamic. This means that class attributes can be added and also modified outside of the classes. This also involves the issue of duplication. In order to avoid this problem, we will create a Python dictionary when processing code files. This allows us to track the class attributes and avoid recording the same attribute names repeatedly.

As shown in Listing 3.11, `Class1` has 6 attributes. Among them, `field3` appears twice in Line 4 and Line 8. Because both of them are attributes belonging to `Class1`, they belong to the same identifier. The same goes for another attribute `field4` in Line 5 and Line 12. The situation of `field5` and `field6` is similar. Both of them are not attributes added when the object is created. Through the dictionary mentioned above, we can prevent them from being repeatedly added to the identifier name table when they are reassigned in Line 16 and Line 17.

4
Results

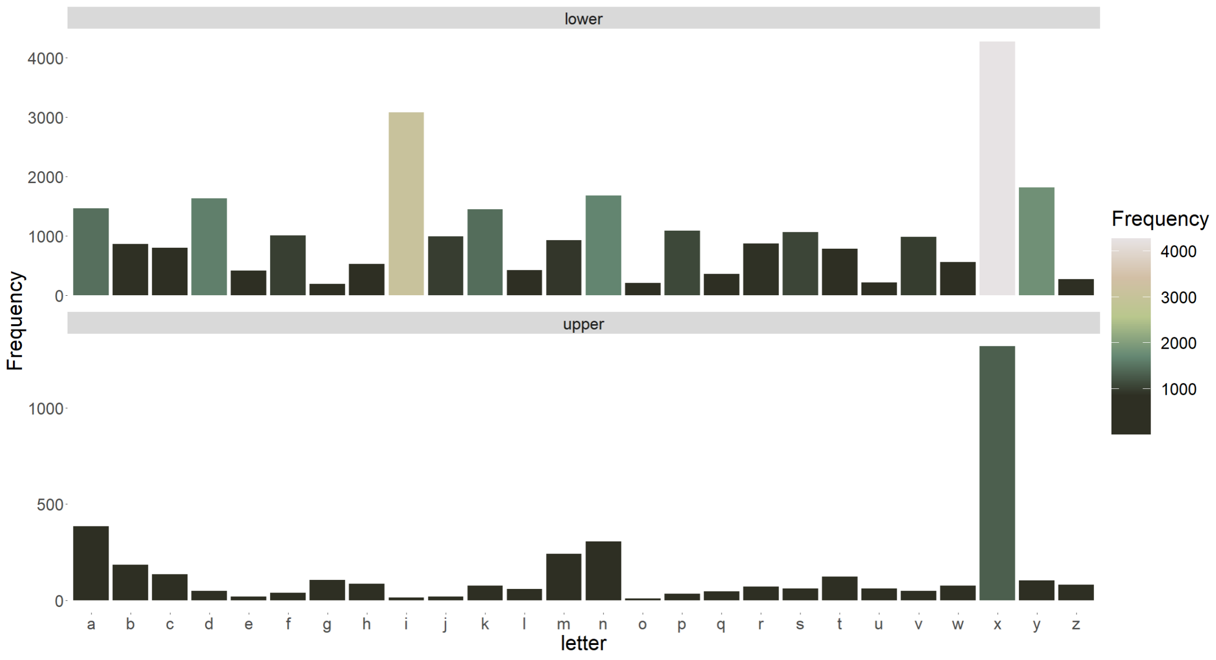


Figure 4.1: Single letter

In this chapter, we are going to present results about the identifier names in Python repositories in a variety of formats including, but not limited to, tables and charts. We not only show how the 9 naming practices are existing in the source code, but also determine some possible motivations through context and repository categories. The results are shown in Appendix 1.

4.1 Single letter

Among all the 745,651 identifiers we investigated, there are 31,741 identifiers that use a single letter as names. The percentage is 4.26%. As shown in Figure 4.1, all of the uppercase and lowercase letters are used at least once. Among them, function/method and class identifiers are rarely named with a single letter, which are 0.15% and 0.0074% respectively.

In order to understand in what particular cases single letter is used as a class name, we need to discuss the specific ones. We found there are only 2 classes are named by a single letter. Both are extracted from `Django` and use `F` and `Q` as class names. `Q` is used as an instance of query filters in the database-related files. Although it is possible to associate `Q` with query after having some knowledge of database, we think it is difficult to associate `Q` with filters before reading the comments. It might be easier to understand by adding filter-related words to this identifier name.

The parent class of `F` shows that it is an object that can perform combination operations and is used to construct database query expressions. We think `F` can be one of the

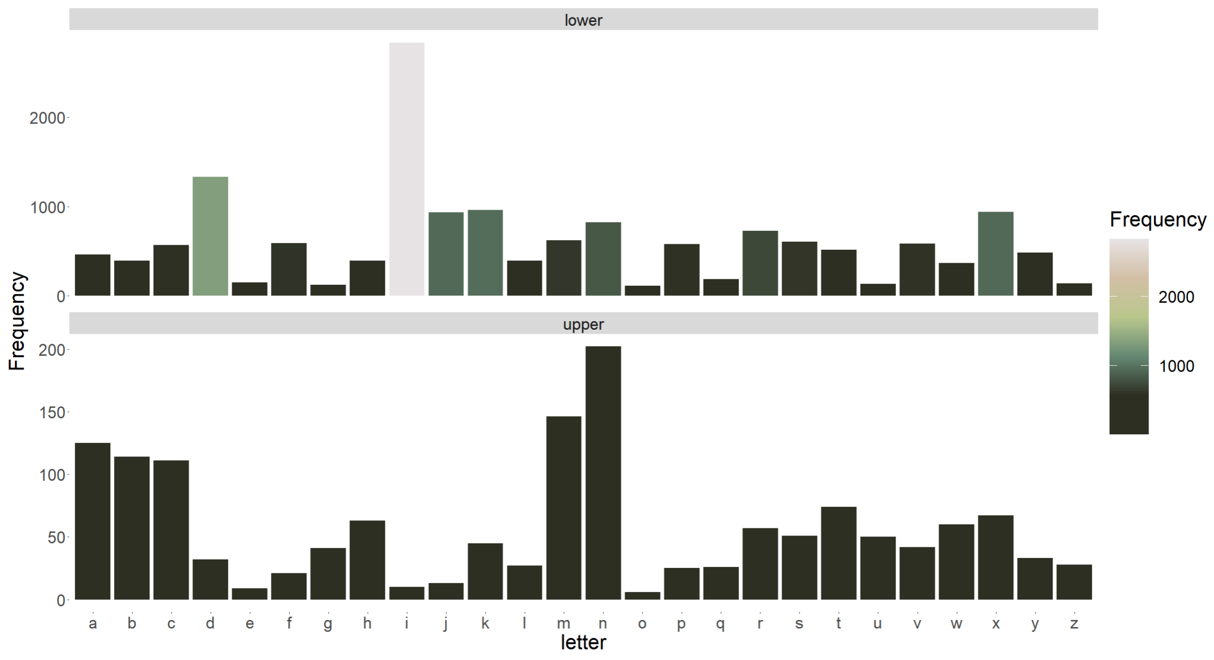


Figure 4.2: Single letter - variable

database-related concepts such as filter or field, which is ambiguous. We believe that the amount of information involved in a single letter is not enough for describing the concepts of classes.

In contrast to class names, the use of single letter is more common in variables, parameters and attributes, where 31,575 of the 31,741 single letter names are these. The percentage of all names of these 3 kinds is 5.21%. In Gresta et al.'s study on identifier names in object oriented programming [24], they investigated the 3 kinds of identifier names, where the percentage of single letter names is 4.33% in C++ and 9.83% in Java. Moreover, in Beniamini et al.'s study [5], they only investigated the percentage of single letter names usage in variable was analysed. The usages are 14% for C, 9% for Java, 20% for Perl, 4% for PHP and 37% for JavaScript. Whereas in our study, the usages are 5.77% for variables , 5.49% for parameters and 0.92% for attributes. This indicates that using single letter names in Python is less common compared to these programming languages which are C, Java, Perl and JavaScript.

The most frequently used letters shown in Figure 4.1 are `x` and `i`. Not surprisingly, the lowercase `i` is used as an index, which is an accepted practice among programmers. Moreover, `j` and `k` are also commonly used in the same way as `i`. The use of `x` is different, as none of the other five languages in Beniamini et al.'s study of variable showed any significant use of `x` as an identifier name [5]. This then makes it necessary to discuss variables and parameters separately. Even though the usage of single letter name is similar in terms of the percentage of variables and parameters, the actual usage is quite different which is shown in Figure 4.2 and Figure 4.3.

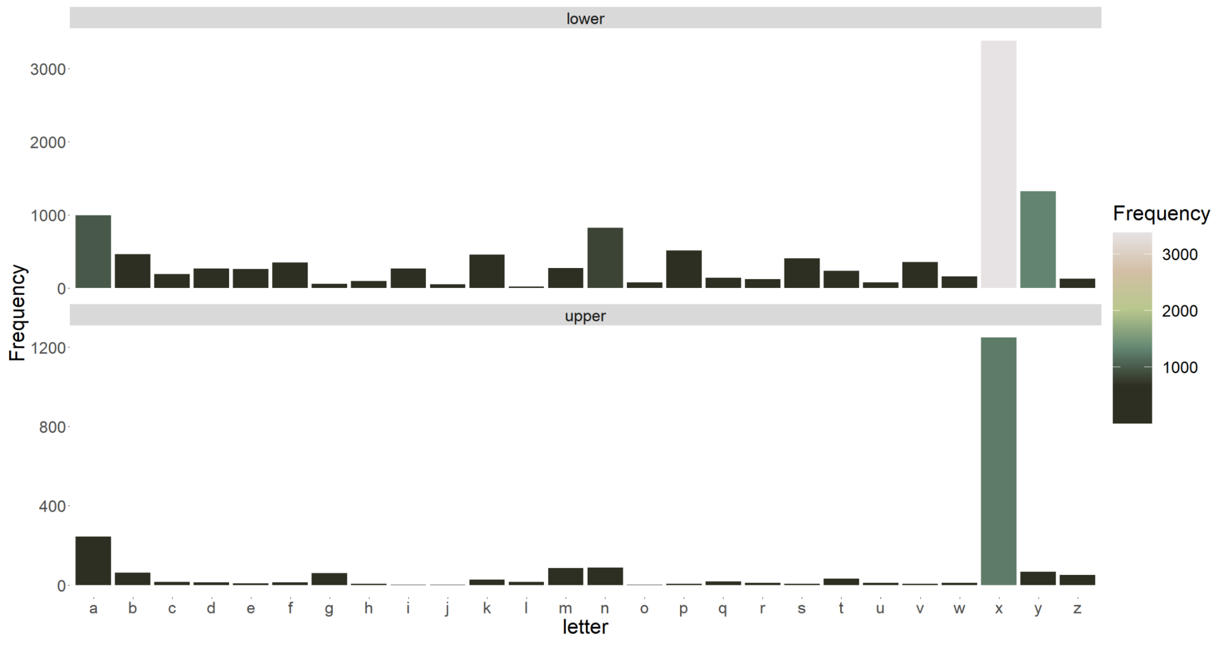


Figure 4.3: Single letter - parameter

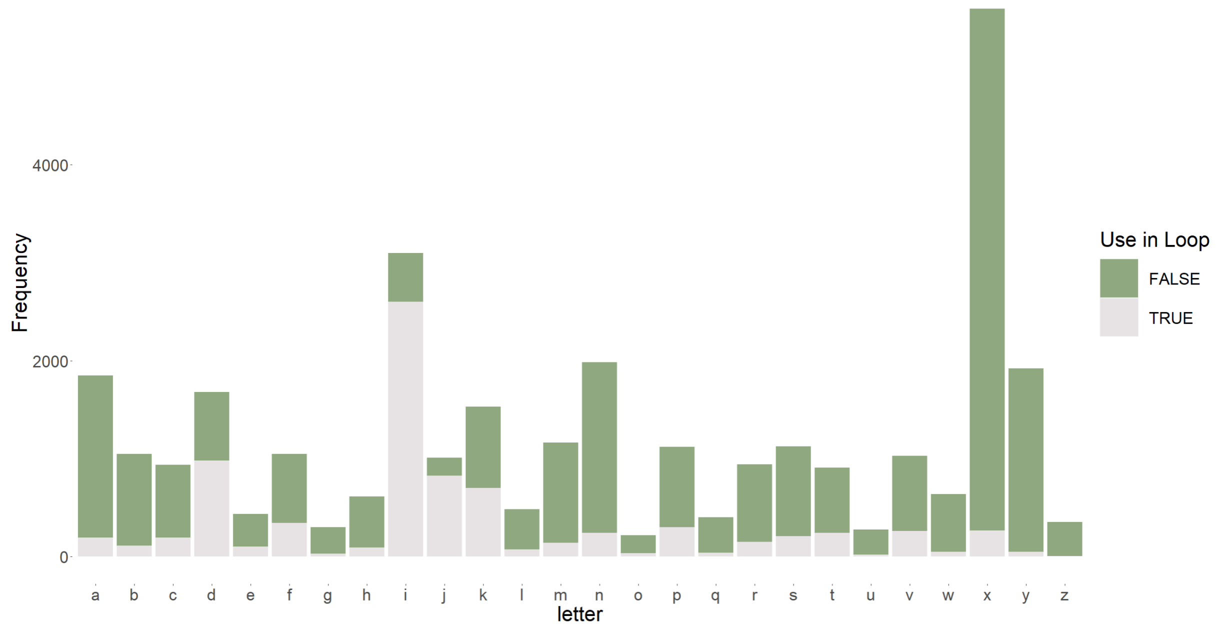


Figure 4.4: Single letter used in For loop

As shown in Figure 4.2, the use of `x` as a variable name is also relatively uncommon compare to the usage of `x` in all the identifier names in Python. About $\frac{3}{4}$ of the `x's` are parameters. This is similar to the results in other languages [5]. Figure 4.4 shows the usage of all single letters as identifier names which is divided by the use of For loop statements. The white bars refer to single letter names used in For loop statements, and the green ones refer to single letter names are used for other purposes. As shown in Figure

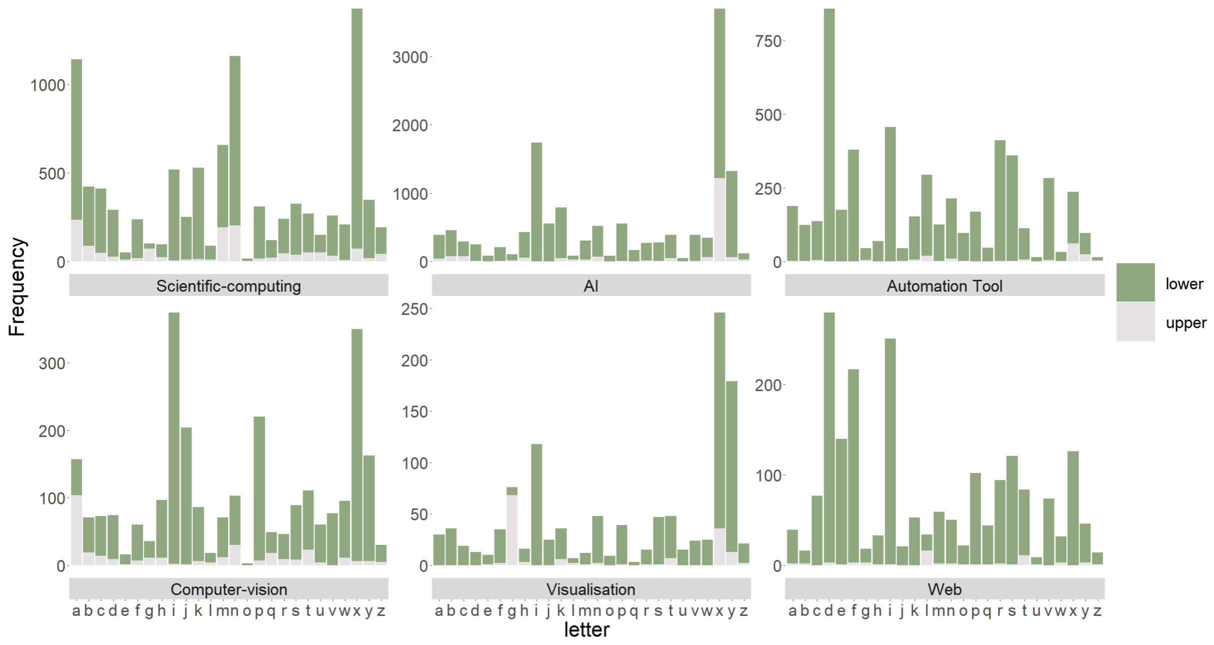


Figure 4.5: Single letter - repository category

4.4, For loop statements are the main use of single letter names `i`.

As our results shown, letter `x` is mostly used to name parameters. In the survey by Beniamini et al. [5], the participants thought that `x` is mostly used for coordinate and math operation and another study mentioned that in Java, `x`, `y` and `z` are well known abbreviations which can be used to represent any numeric types [10]. In order to confirm whether this statement also applies in Python naming, we investigated how `x` is used in different categories of repositories (Figure 4.5). We noticed that `x` is relatively less often used as identifier names in Web and Automation tools than in other types. This may be due to the fact that they are not really associated with coordinates or math operations and the small number of `x` applications may come from repositories that have more than 1 category. For example, `ralph` (a visual data management system) is associated with both Web and Visualisation. Moreover, the source code shows that most of the `x` applications in scientific-computing are related to math operations, while computer-vision, on the other hand, is usually related to coordinates.

In addition to the use of `x`, Figure 4.5 also shows that there is certain usage of single letter names in different categories. The similarity between the automation tool and the web chart is due to the fact that there are 11 repositories in both categories. We can still notice that in web-related repositories, the use of `f` as an identifier name is relatively more frequent than other categories. Moreover, `d` is commonly used in both Automation tool and Web, but rarely seen in other categories. Other cases are the lowercase `a` in scientific-computing and the uppercase `A` in visualisation.

```python
for f in fields:

for f in forms:

for f in temp_files:
```

Listing 4.1: Letter f used in For loop

In Web-related repos, 44% of `f`'s are used as loop variables and 41% as parameter, which account for 85% of all. In the case of using as loop variables, all the letters `f` appear in the list type For loop, which is shown in List 4.1. These 3 examples are from Web-related repositories. Letter `f` is also commonly used to abbreviate `field`, `form`, and `file` in these cases. We believe that the concepts of acronyms in a small scope might seem obvious, but it can still have a negative impact on comprehensibility. Even though the 3 abbreviations for `f` in Listing 4.1 exist in several different repositories, they are all web-related words. This means that programmers might have multiple different interpretations when they notice a letter `f` in the code. We believe using full words, such as `field`, `form` and `file`, is a better choice.

There are also some other cases where it is possible to tell the concepts of the single letter names from the context that it is being passed as an abbreviation for a function, but there are also several cases in the code where it is difficult to tell the meaning just by reading the function itself. We believe this has a negative impact on comprehensibility since it might be a general practice that requires some knowledge of the repository in order to understand it.

The letter `d` is also used in list type For loop statements, which is similar to the cases of `f`. Beyond that, it usually refers to data-related concepts such as `data`, `document`, and `dictionary`. There are some exceptions, such as using `d` as `distance`.

```python
def cmp(a, b):
    return (a > b) - (a < b)
```

Listing 4.2: Letter a used in computational function

In addition to the list type For loop as an abbreviation for any list concepts, we find two common cases where the lowercase letter `a` is used. One is as an abbreviation for array. Another is as a parameter, which is commonly used in some computational functions as a float number or vector (Listing 4.2). Both of the cases are related to scientific-computing, so we believe that the use of such abbreviations in such specific contexts is acceptable. But we think it is still important to be careful not to use the same acronyms to name different concepts in certain scopes.

The letters `a`, `d` and `f` mentioned in the previous paragraphs are used as variables in list type For loop statements, while most of the other letters have similar applications.Figure

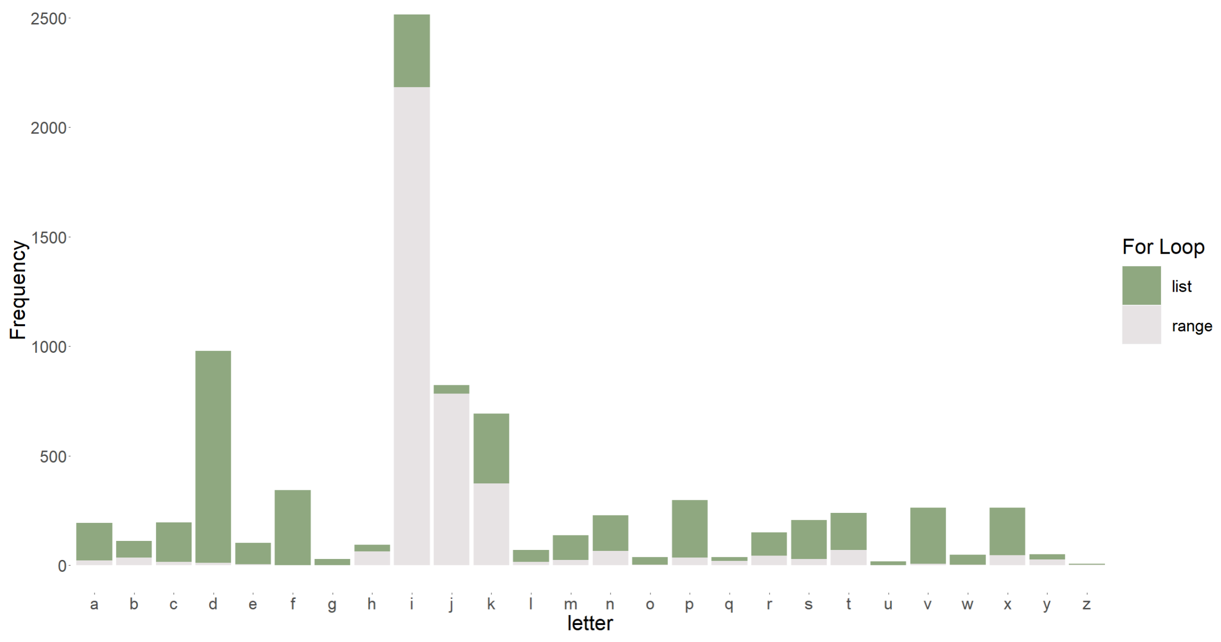


Figure 4.6: Single letter - For loop

4.6 shows that the letter `i` is the most used letter in For loop statements. The letters `i`, `j` and `k` are mostly used in range type For loops because they are recognised by programmers as identifier names that can be used for indices. The other letters are mostly used in range loops, and their main use is as abbreviations for the concepts represented by the lists in list loops. Moreover, uppercase letters are almost never used in For loop statements, which is probably related to the Python naming styles, where variables usually start with a lowercase letter.

We also investigated the use of single letter identifiers in While loop statements which account for 25.05% of all variables used in While statements. This is very similar to the 27.19% for For loop statements. However, in contrast to For loop statements, not all the single letters are used in While loop statements (Figure 4.7). As shown in Figure 4.8, only 11 kinds of single letters are used. There are maybe two reasons. The first is because, unlike list type For loops as shown in Listing 4.1, there is no notion of allowing abbreviations in While loop statements. Secondly, While loop statements are not as common in the source code. In our data set there are 8,140 single letter identifiers in For loop statements, but only 132 in While loop statements. However, since the percentages are similar, we can still conclude that these single letters, which are not used, are rarely or only used in specific cases for While loop statements. Similar to For loop statements, single letter `i` is also commonly used in While loop statements which also relates to the use of indices. This is also similar to the conclusions of Gresta et al. for Java and C++ [24], where indices are more commonly used in small scopes like loop statements.

We compared whether the size of lines of code (LoC) has an effect on the use of single

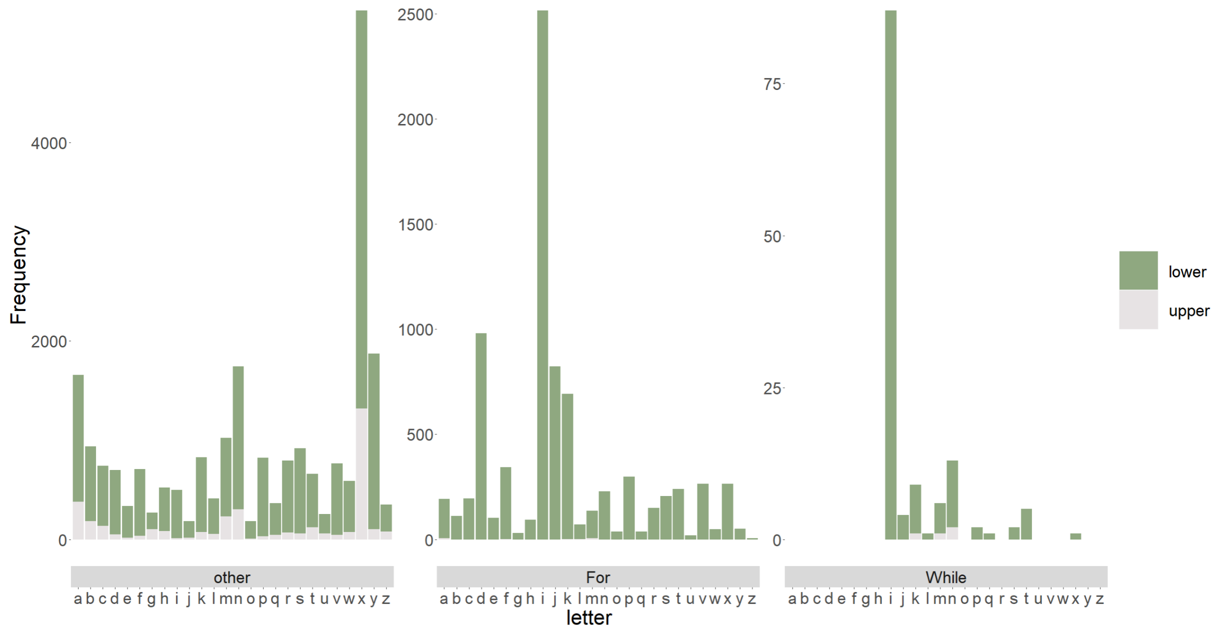


Figure 4.7: Single letter - Loop

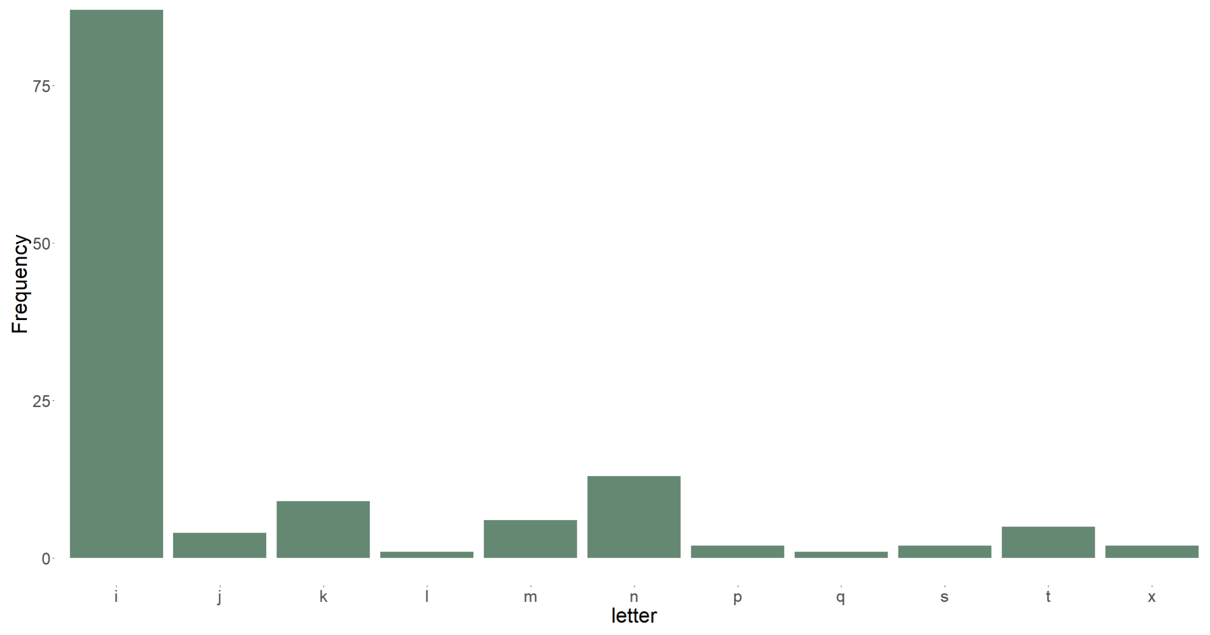


Figure 4.8: Single letter used in While loop

	LoC	Commits	Contributors	Releases	First Release
cor	0.0858	-0.0863	-0.00796	-0.0208	0.0979
p-value	0.4	0.4	0.9	0.1	0.8

Table 4.1: Single letter - Spearman correlation

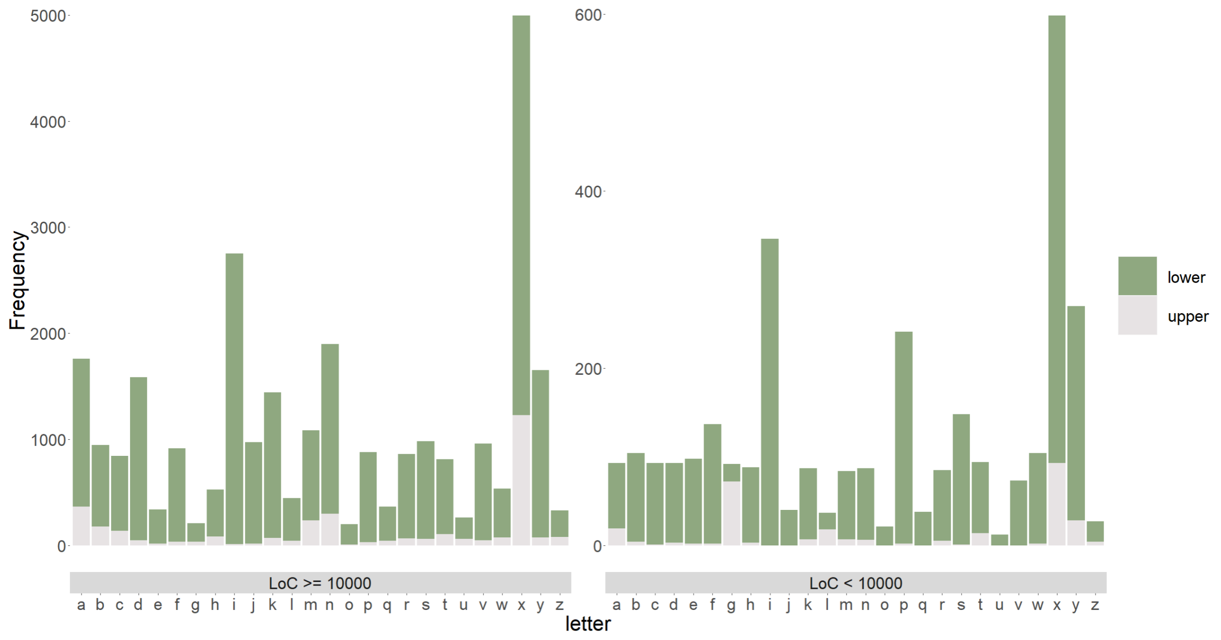


Figure 4.9: Single Letter - LoC

letter identifiers. Overall, single letter identifiers have a slightly higher percentage in smaller repositories. The percentage in small repositories is 4.66% compared to 4.20% in large repositories. The difference is not significant (Table 4.1). Figure 4.9 shows the use of different single letter identifiers. The overall trend is similar. We believe that some of the differences might be related to repository categories. For example, `p` is more commonly used in computer-vision and web repositories (Figure 4.5). Six of the 10 repositories related to computer-vision are small repositories and 19 of the 26 repositories related to the web are so. The use of single letter `a` in large repositories is similar which is related to Scientific-computing. We also investigated whether the number of contributors and commits affects the use of single letter identifiers. The results also show that there is no significant association (Table 4.1).

The use of single letter identifiers varies across repositories (Appendix 1). Among all the 100 repositories used in this thesis, 3 of them have more than 20% of single letter identifiers, namely `corona` (23.08%), `scipy` (21.71%) and `pyxel` (20.00%). The first 2 of them are related to scientific-computing. There are 7 repositories that do not use any single letter to name the identifiers. This might be due to their size. Six of these repositories have LoC less than 400. Excluding the 6 repositories, there are 14 others that use less than 1% of single letter identifiers. Eight are related to web and 9 are related to automation tool. Thus, we believe that the use of single letter identifiers is related to repository categories.

4.2 Number at end

Out of all 745,651 identifier names investigated in this thesis, there are 12,416 identifier names that end with a number. They account for 1.67% of all identifiers involved in this thesis. Among these, there are 871 (0.73%) identifier names ending with a number used for naming functions/methods, and 223 (0.83%) names used for naming classes. Of all the 100 repositories used in this thesis, there are 77 have this naming practices.

For this kind of names used to name classes, the uses of these numbers can be divided into 3 types. They are distinguishers, versions and specific uses which have been discussed in Section 3.2.

The first is distinguishers. It is used to differentiate between specific functions when they have the same concepts. An example is the two class identifiers `nhwc3to4` and `nhwc3to8` in `AITemplate`. They are used to convert 3-channel data into either 4-channel or 8-channel. The first half of the abbreviation is not discussed here for the moment. We think it is reasonable to use numeric endings in this case. The two identifiers have nearly the same concepts. It is also easier to read than using the full word version `three_to_four`.

Secondly, there are some specific words that end in numbers, such as `web3` and `Jinja2`. We think it would not be a problem if these words existed in programmers' knowledge base.

The third is a suffix that is used to refer to the version of the class. This is used a lot in `Gymnasium`, such as `ResizeObservationV0` and `NormalizeRewardV1`. However, in all the repositories we used, there are no cases where both versions of the class occur at the same time. They all keep only the latest version, so there is no distinction to be made. Therefore, we consider this suffix to be redundant. As mentioned in Section 3.2, we believe that version information should be stored as comment.

We also investigated the practical application of this type of names in functions and methods. Except the auto-generated number, all the 3 types of number suffixes are used in both functions and methods and we found that there is no difference in the use of names ending in numbers between functions and methods. In contrast to class names, specific uses of numbers are more common in functions/methods. For example, `read_float64` is a function used to read 64-bit float numbers. 64-bit related names are common in the category number in middle [23], but we find it used in number at end as well. Moreover, we investigated functions and methods separately (Figure 4.10). The name `read_float64` does not become more easier to understand by using the full word version `read_float_sixty_four`. The same applies to other identifier names such as `rot90` in `albumentations`.

We found that there are cases where using multiple names with the same prefix but with numbers belonging to special purposes as identifier names for functions/methods

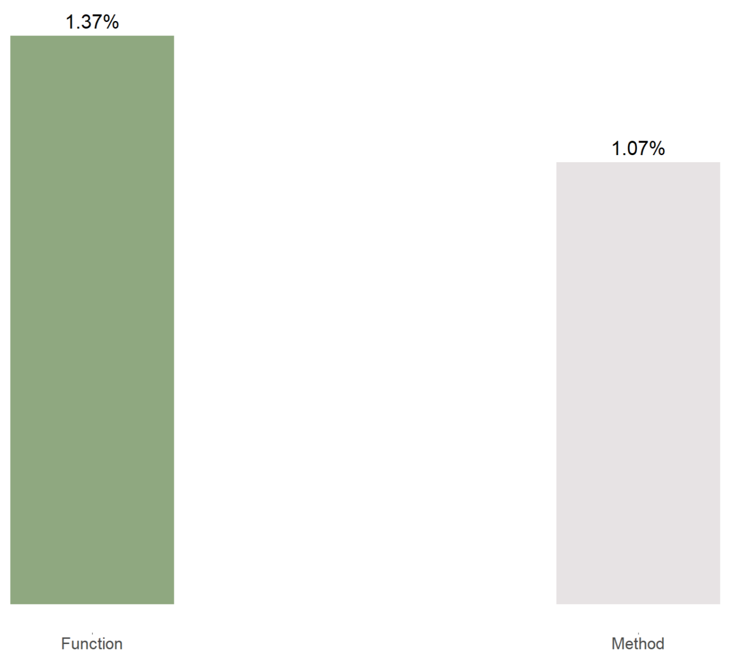


Figure 4.10: Number at end - function and method

may cause another code maintainability problem which is alterability. The bodies of these functions/methods are usually highly repetitive.This is because they refer to similar concepts and the functionality is also similar. For example, there are 14 such methods in `numpy` which are `_is_r2000`, `_is_r3000`, `_is_r3900`, etc. All of these methods perform exactly the same function of processing and determining CPU information and the suffix stands for the CPU info. The only difference is the output, which is also related to the number suffix. The problem here is that if the functionality of one of the methods needs to be modified, then this step has to be repeated 14 times. The alterability is very low. We think it is impossible to change these incomprehensible identifier names without refactoring the code.

In contrast to function/method and class names, the use of names ending with a number is more common in variables, parameters, and attributes. There are 11,376 which accounts 1.87% of all the identifier names for variables, parameters, and attributes. This is less common compared to Java (20.79%) and C++ (7.82%) [24]. The data for Java [24] may be affected by 2 large repositories which are greater than 40%. After removing these two extremes, there are still about 9.76% of variables, parameters and attributes are named by identifier names ending with a number. This is still higher than the Python results in this thesis. We think this might be related to the different uses of different programming languages, as mentioned earlier in Section 3.3.

We investigated the use of variables, parameters, and attributes where the scope is functions/methods.Figure 4.11 shows the percentage of identifier names that use names ending in numbers in functions/methods of different lengths. The length is the number of

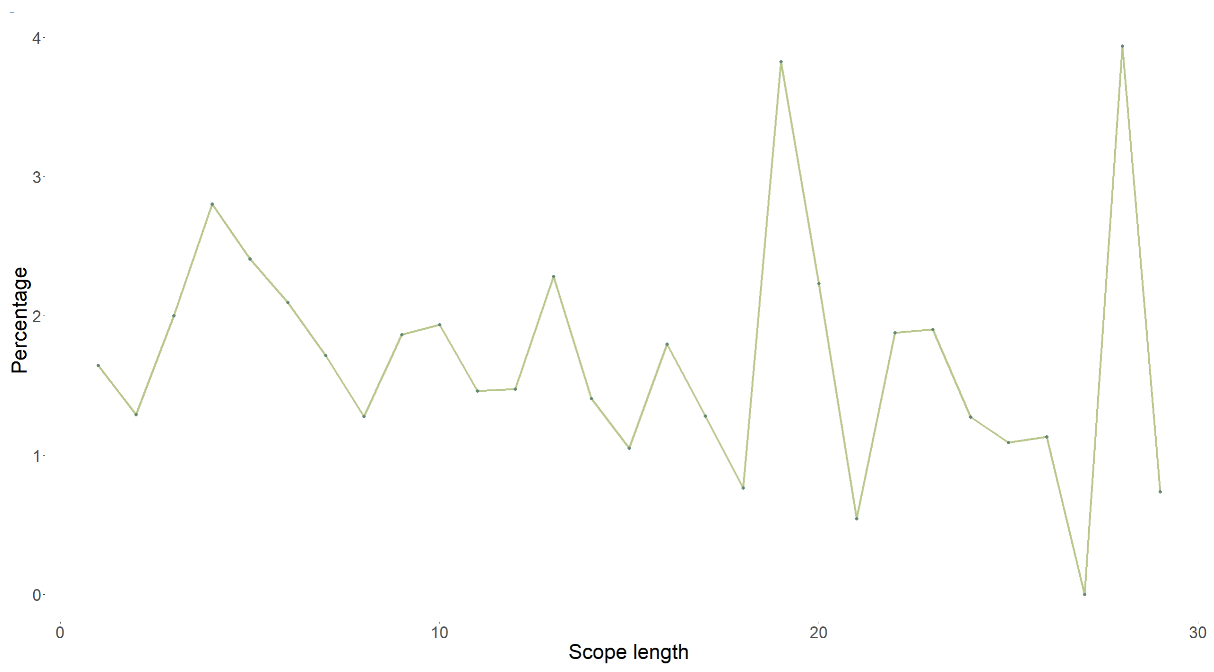


Figure 4.11: Number at end - function/method scope length

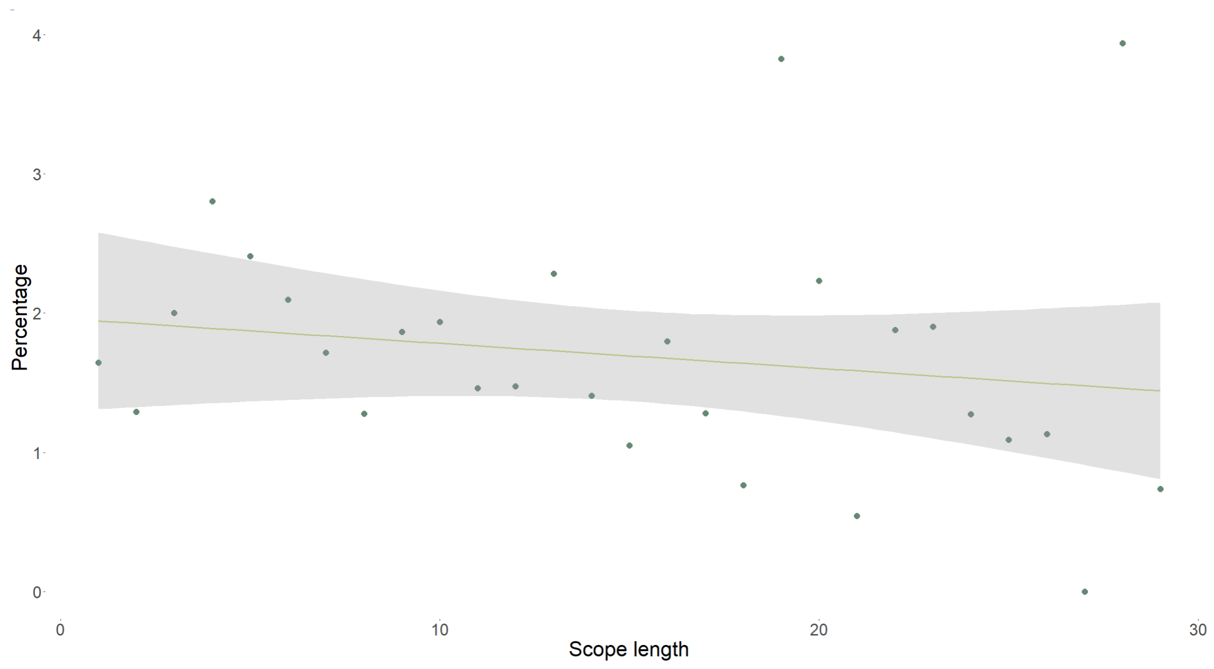


Figure 4.12: Number at end - function/method scope length

lines of code of the function/method. Since 99% of the identifiers are in functions/methods which have less than 30 lines, we only plot this part. We tested the Pearson correlation coefficient. The result is that correlation is -0.182 and p-value is 0.3. Therefore, the data does not show linear relationship.

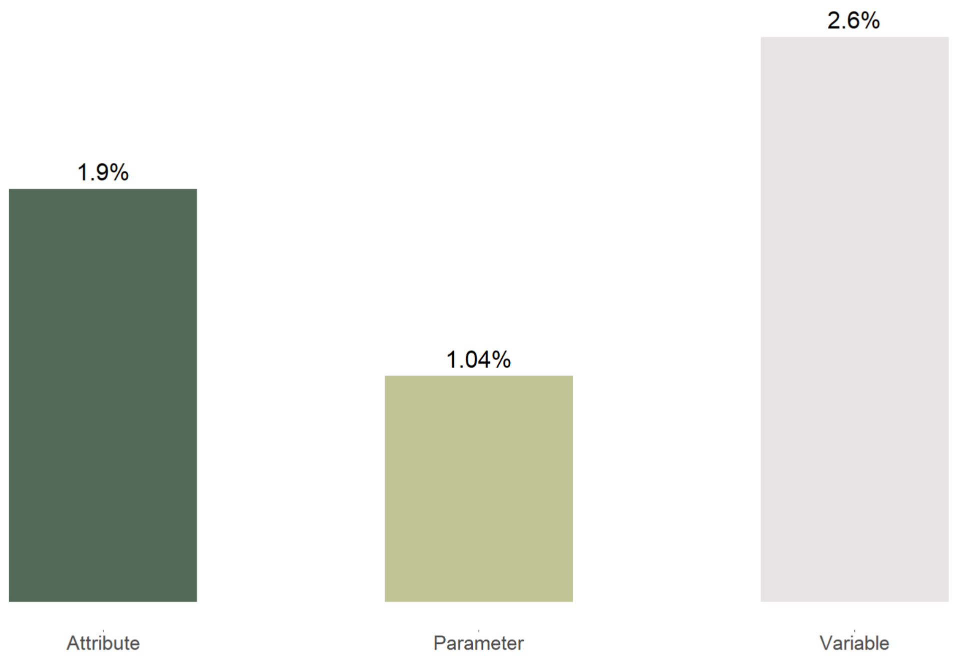


Figure 4.13: Number at end - variable, parameter and attribute

```
def get_broadcast_max_shape(shape1, shape2):

def intersection(boxes1, boxes2):

def forward(self, p2, p3, p4):
```

Listing 4.3: Number at end used as parameters

We also discuss variables, parameters, and attributes separately. As shown in Figure 4.13, identifier names ending with a number account for 1.66% in attributes, 1.03% in parameters and 2.56% in variables. Compared to the other two identifier names, the use of such names in parameters is the least. Although it has a small percentage, identifier names ending in numbers used for parameters are almost used as distinguishers. They usually appear as a plural number of the same prefix combined with a sequence of numeric endings (Listing 4.3). Moreover, 54 of the 77 repositories that contain identifier names of this naming practice have occurrences of this kind of application in Listings 4.3. Refer to Section 3.2, we consider this application to be a consensus, and doing so in a small scope would not cause comprehensibility problems. The results also shows that the use of this type of parameters are similar in functions and methods. Moreover, the uses of number at end in attributes and variables are similar. In addition to the common use in parameters to provide some distinction between similar concepts, there are also suffixes indicating 64-bits or 95% percentile. We consider these to be acceptable identifier names since these numbers are only used when necessary.

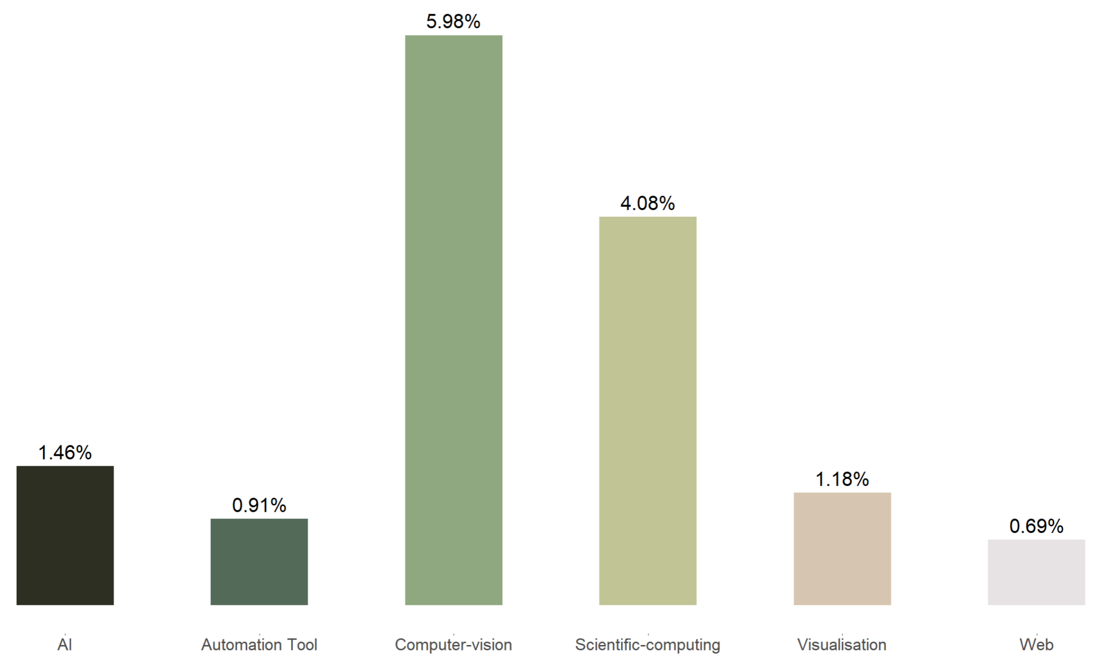


Figure 4.14: Number at end - category

```
for c1 in self.standings:
    for c2 in self.standings:
```

Listing 4.4: Distinguishers used in Loop

Using names ending in numbers is less common in Loop compared to 1.67% of all identifier names. The percentage is 0.5% for usages in For loop statements and 0.78% for usages in While loop statements. The most common use is distinguisher which is used to distinguish between different levels of loop variables in a nested loop. For example, as shown in Listing 4.4, the identifiers `c1` and `c2` are used for this purpose in `erpnext`.

As mentioned above, we believe that different uses of programming languages affect the use of numbers in identifier names. We also investigated the use of number at end in different repository categories. As shown in Figure 4.14, identifier names ending with a number are more common in computer-vision (5.98%) and scientific-computing (4.08%) than in other repository categories. This might be due to the characteristics of the categories. Firstly, in computer-vision repositories, there are a large number of uses related to the axes and points. For example, `x1`, `y1` and `p1`. In scientific-computing, we notice that there are many identifier names related to algebra and vectors. For example, `x1`, `a1`, `r1` and `vector1`. These identifier names mentioned above can also be categorised as providing distinction between similar concepts. As mentioned in Section 3.2, we believe that the reason for the relatively small proportion of such names in other repository categories might be because they are less likely to have identifiers with similar concepts to distinguish.

	LoC	Commits	Contributors	Releases	First Release
cor	0.459	0.194	-0.0109	-0.0922	-0.266
p-value	$2\text{x}10^{-6}$	0.06	0.9	0.5	0.4

Table 4.2: Number at end - Spearman correlation

The use of number at end varies in different repositories. Even though the average value is 1.67% out of 100 repositories, there are still 3 repositories where names ending in numbers account for more than 5% of the repositories. These are `eht-imaging` (12.24%), `spaCy` (7.63%) and `SiamMask` (6.55%). They are all related to Computer-vision or AI, which is in line with the results presented in Figure 4.14. There are 23 repositories which do not use such identifier names at all. We think this may be related to that they are all small repositories with LoC less than 5,000. We investigated 15 repositories other than these 23 that had a number at end usage of less than 0.4%, and found that 8 of them are related to the Automation tool, and 7 are related to the Web. Only 1 repository is related to computer-vision, and none of them are related to AI or Scientific-computing. This again confirms that different categories have an impact on the use of number at end. There are also similarities across repositories in that there are almost no auto-generated meaningless numeric endings seen in any of the repositories. We believe this may be because these names are mostly found in test files and other files excluded from our thesis. Certainly the use of such names is harmful for comprehensibility. Therefore, we believe that the repositories we investigated are not problematic, at least in terms of this aspect of identifier name choice.

As shown in Table 4.2, we investigated the correlation between properties of repositories and the usage of number at end. By Spearman correlation, we found that commits, contributors, releases and first release dates all have no correlation with the percentage of identifier names with a number at end. This is the same as in Java and C++ repositories [24]. However, in Python, using names ending in numbers tends to be more common in larger repositories (correlation = 0.459).

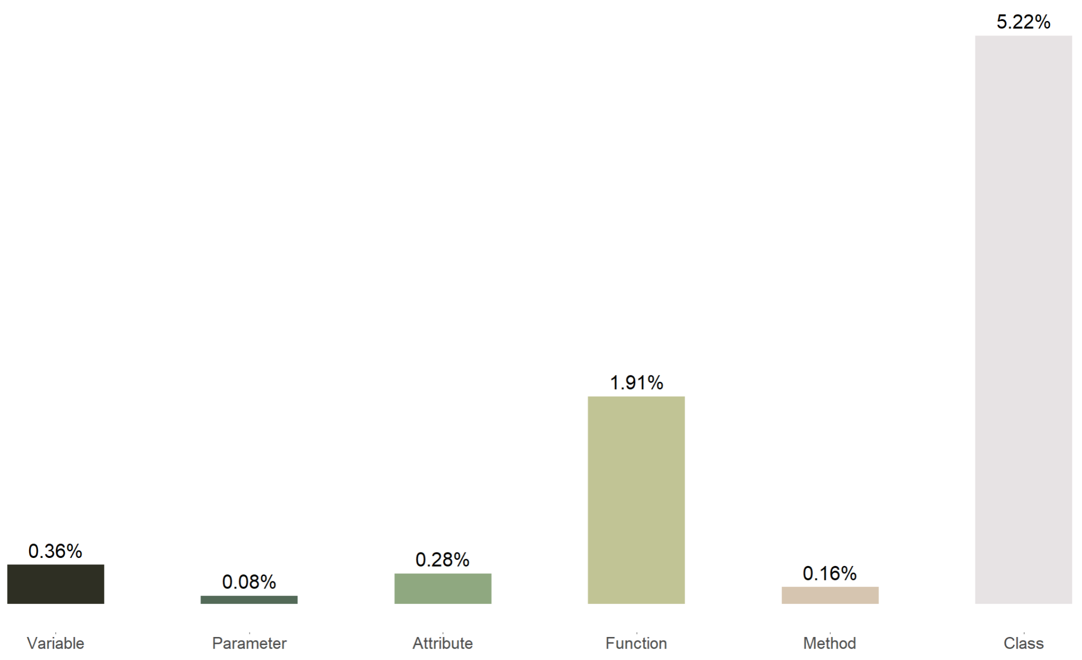


Figure 4.15: Number in middle - Percentage

4.3 Number in middle

Number in middle is less common in Python source code compared to other naming categories discussed in this thesis. In all the 745,651 identifier names, only 3,643 of them belong to this category. They account for 0.49%. Among them, 175 identifier names belong to both number at end and number in middle. This is 4.80% of all the identifiers belong to number in middle. Most of these overlaps are used as suffixes. For example, as mentioned in Section 4.2, the two distinguishers `nhwc3to4` and `nhwc3to8` in `AITemplate` are belonging to both of the naming categories. Another case is `Matrix3x3` which is used in `manim` to refer to the size of the matrix. We think such identifier names should belong to number at end. Since we have discussed these in Section 4.2, we are not going to repeat the discussion in this section.

As shown in Figure 4.15, the use of number in middle varies across different kinds of identifier names. It accounts for 5.22% of all class names. 0.72% in functions/methods. 0.23% in variables, parameters and attributes. The use of number in middle is less common than 7.65% in Java and 3.27% in C++. [24]. Moreover, the usage in Python is 0.36% in variables, 0.08% in parameters and 0.28% in attributes.

We investigated the use of number in middle for class names. We found that there are 3 types of class names named after the class name, synonyms, specific uses and versions which have been discussed in Section 3.2. Firstly, for the synonyms, it is usually used to replace monosyllabic words. For example, the number `2` in `Speech2TextDecoder` in `FlexGen` is used to replace the word `to`. Therefore, the full word version of this name is `SpeechToTextDecoder`. We think that the use of numbers in this case would better emphasise that speech and text are the main concepts intended by the class name. It separates the words `Speech` and `Text` by the number `2` to make them seem more obvious. Interestingly, we found that synonyms only exist when the naming style is camelCase. It is never used in under_score style names. This might cause by that the words `Speech` and `Text` seem more obvious in `Speech_to_text_decoder`.

The other case is specific uses. For example, `LayoutLMv3Model` in `FlexGen` and `Md5File` in `pyinfra`. LayoutLMv3 is a pre-trained model used for document and md5 is a widely used message-digest algorithm. As mentioned in Section 3.2, this kind of identifier names are meaningful, since the numbers are required for the certain concepts. Furthermore, for class names, most of the number in the middle are specific uses. They are basically used for concepts relate to dimensions, such as `3D` stands for 3-dimensional. This will be discussed in detail in category paragraph.

For the number used as versions, it is only been used in the repository `FlareSolverr`. It used `V1RequestBase` and `V1ResponseBase` to name 2 class names. We have not found any other versions of the classes in this repository, such as `V0` or `V2`. Moreover, they also provided some version-related information as comments in the class body. As mentioned in Section 3.2, we think it would be better to replace these identifier names with `RequestBase` and `ResponseBase`.

In contrast to class names, the use of this category of identifier names is less common in functions/methods. These function/method names are used for two purposes: synonyms and specific uses.

Dimension-related concepts are also the most common uses, such as`conv2d_filter` in `AITemplate`. As mentioned in Section 3.2 and Section 4.2, changing these names to full word form only reduces their readability. We don't think using `twoD` or `two_d` is more obvious than using the word `2D`.

For the synonyms, we found the use of `str2bool` in 5 different repositories. Other uses include `str2int`, which converts a String to an integer. This may be due to the characteristics of Python. It is a dynamic programming language, and the type of its variables is not fixed. This naming convention does shorten the length of the identifier name. It also makes it easier for the programmers to recognise the keywords when looking at the identifier names. This case refers to the type that needs to be converted. However, using numbers in middle of the names are difficult to search for in code and difficult to pronounce [39]. We believe that if this kind of names are required, all similar uses of the word `to` in a repository should be replaced with `2`. Otherwise, do not use it at all. This is because mixing `2` and `to` may cause confusion. For example, when a programmer searches for the function it may not be clear whether `strToInt` or `str2int` should be used, which would reduce the efficiency of maintaining code.

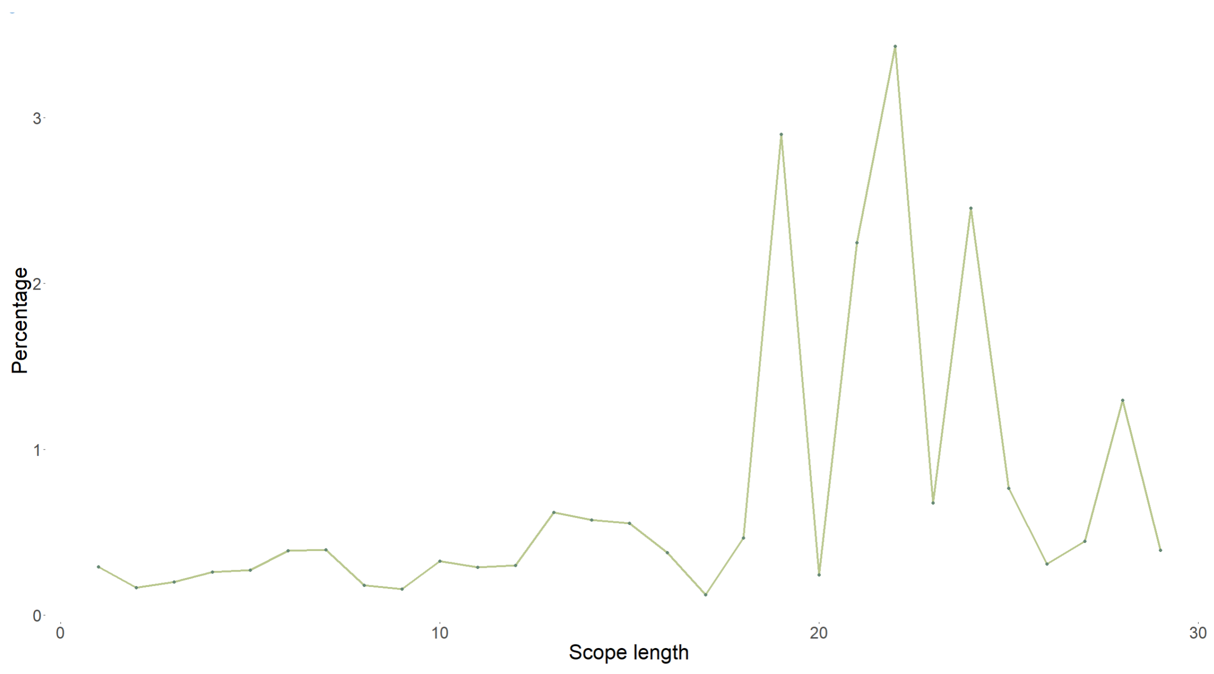


Figure 4.16: Number in middle - function/method scope length - attribute & variable

Unlike parameters ending with a number, which are mostly distinguishers, we found that parameters with numbers in the middle are mostly used as synonyms and specific uses. This is similar to other identifier names. For synonyms, we also found cases where `2` is used to replace the word `to` in parameters. Among these identifier names, we found some of them have comprehensibility issues, but we think they are not very relevant to the numbers themselves. For example, `w2h_ratio` in `albumentations` and `m2m_data` in `django`. Both of these identifier names use `2` instead of the word `to`. As mentioned in Section 3.2, if their use is consistent throughout the repository, there is nearly no impact on searchability or pronunciation. In the two cases, it is the letters `w`, `h` and `m` around number `2` that cause the comprehensibility issues. As shown in the comments, `w2h` is an abbreviation for width to height and `m2m` is an abbreviation for many to many. We think it is difficult to understand the meanings from the names themselves and the background information of the repositories without reading their comments. However, using `width2height_ratio` and `many2many_data` would make these identifier names more easier to understand. Similar to other kinds of identifier names, names with numbers related to dimensions are also the most common ones in parameters.

By using Pearson correlation, we found that the use of variables and attributes in functions/methods is positively and linearly correlated with the length of the scope across the 100 repositories investigated. The correlation is 0.402 and the p-value is 0.03. Figure 4.16 demonstrates the percentage of the occurrence of number in middle of the type of identifier names with scope length less than 30. This implies that variables and attributes of the number in middle type are more common in larger functions/methods. In our data

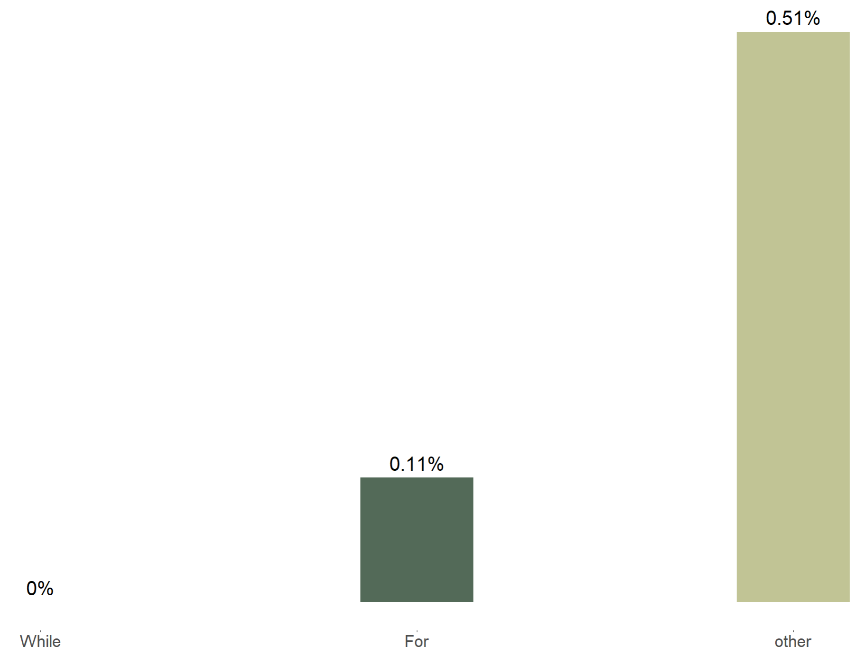


Figure 4.17: Number in middle - Loop

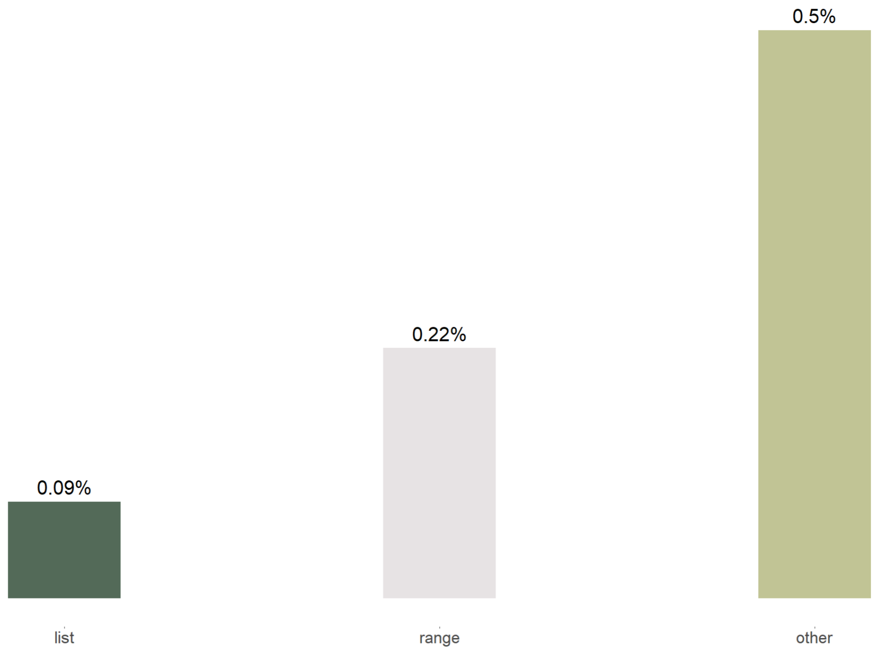


Figure 4.18: Number in middle - For loop

set, we found that there is no significant association between the usages of number in middle parameters and the scope length of functions/methods which they belong to. The correlation is 0.0731 and the p-value is 0.7.

We also investigated the occurrence of number in middle as variables in loop statements. As shown in Figure 4.17, we found that none of the 527 variables used in While loop statements used numbers in the middle of their names. Although this could be re-

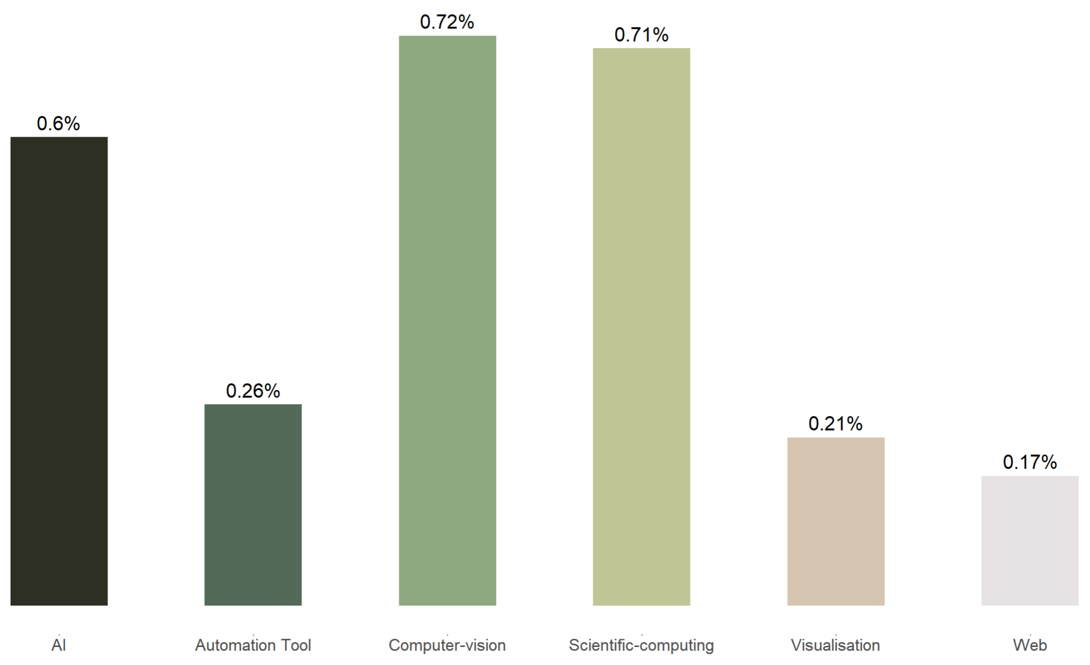


Figure 4.19: Number in middle - Category

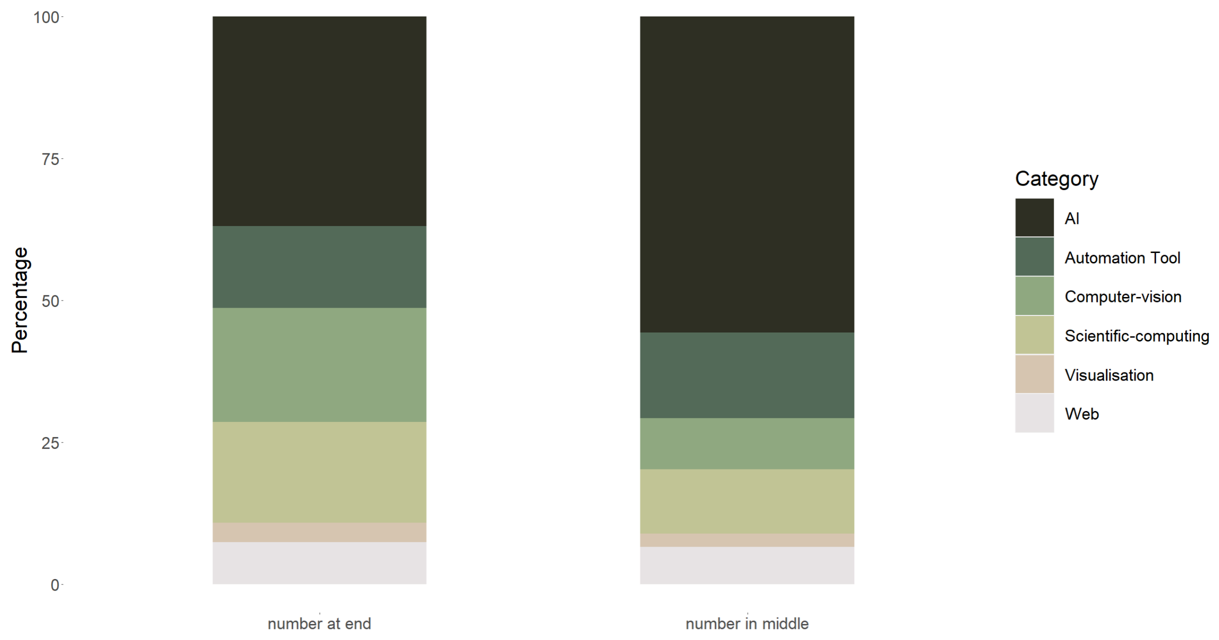


Figure 4.20: Number in middle/Number at end - Category

lated to the bias of our data set, we still can conclude that it is rare to use this kind of identifier names in While loop statements. Moreover, for the names used in For loops (Figure 4.18), we found the purpose of using number in middle is similar to other kinds of identifier names. Moreover, the variables in both list type and range type For loop statements are used as synonyms or specific uses. We have not found it to be special in these cases.

	LoC	Commits	Contributors	Releases	First Release
cor	0.536	0.21	0.104	-0.121	-0.196
p-value	$9\text{x}10^{-9}$	0.04	0.4	0.4	0.5

Table 4.3: number in middle - Spearman correlation

As shown in Figure 4.19, the use of number in middle is different in different repository categories. This kind of identifier names are more common in AI (0.60%), computer-vision (0.72%) and scientific-computing (0.71%) related repositories. This is different from the distribution of numbers at end (Figure 4.20). We believe that naming with this type of identifier names is relevant to repository categories. This is because we found that of all the number in the middle identifier names, regardless of the kinds (variable, parameter, attribute, function/method, class), many of them are related to dimensions. For example, the use of `3d` to refer to three dimensions. This is commonly used in computer-vision and AI related repositories. This is related to the convolutional neural networks which is a deep learning algorithm widely used for analysing visual images. Dimension-related names are also widely used in scientific-computing repositories for represent the dimension of matrix. Another use is to refer to the Mathematical power. For example, the identifier name `x2y` refers to x^2y. This might be reason that there are more this type of identifier names in the 3 repository categories. It does not happen in other repositories.

Overall, different repositories have different applications for using numbers in the middle of identifier names. Firstly, among all of the 100 repositories, this type of identifier names are never used in 39 repositories. This is more than other naming categories investigated in this thesis. It is also consistent with that number in middle is the least common one as mentioned in the beginning of this section. Of the 9 repositories with a percentage greater than 1%, 3 are related to AI and 2 are related to computer-vision and scientific-computing, respectively. Moreover, only one is related to automation tool and none of the repositories are related to visualisation and web. This is consistent with the results shown in Figure 4.19.

We investigated the correlation between the application of number in the middle in the code and the properties of repositories. By the Spearman correlation (Table 4.3), We found that in the data used in this thesis, number in middle are more likely to occur in larger repositories. Since LoC has linear relationship with the percentage of number in middle. The p-value is $9\text{x}10^{-9}$ and the correlation is 0.536. The statistical result also shows that it is more likely to occur in repositories with more commits. The p-value is 0.04. Since the correlation value is 0.21, the relationship is less stronger than LoC. There is no evidence showing that there is any significant association between the usages of number in middle and the other 3 repository properties, that is number of contributors, number of releases and the first release dates.

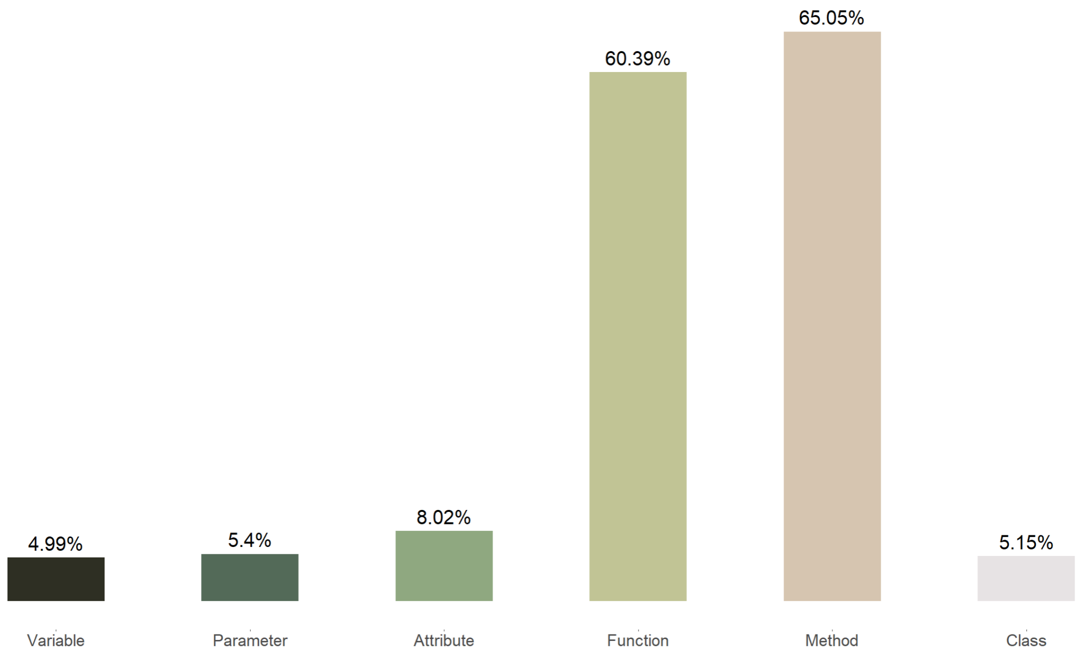


Figure 4.21: Verb phrase - Percentage

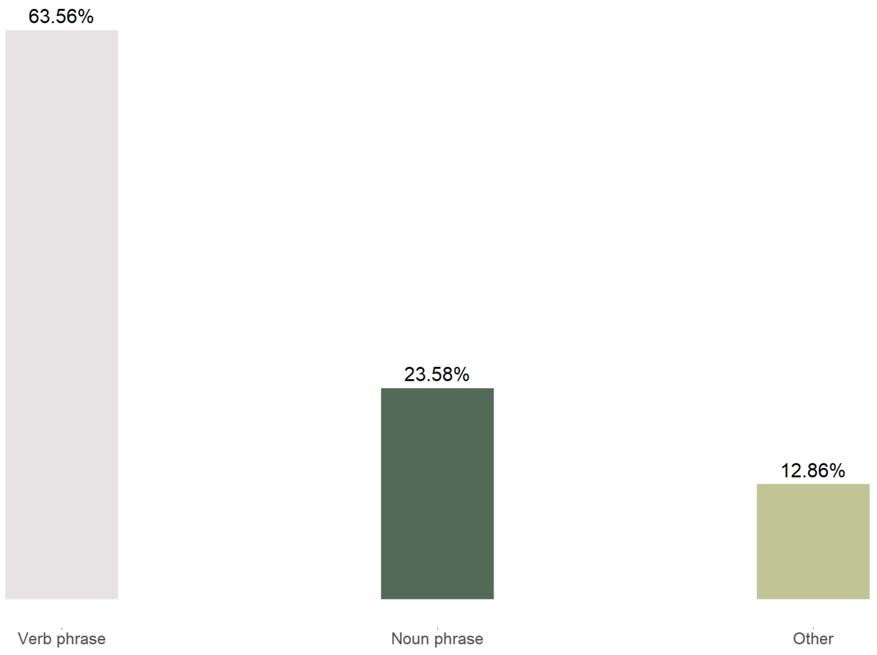


Figure 4.22: Function/method - Percentage

4.4 Verb Phrases for functions/methods

Out of the total 745,651 identifier names, we classified 106884 of them are verb phrases. This accounts for 14.34%. As shown in Figure 4.21, functions (60.39%) and methods (65.05%) have the highest percentage. As shown in Figure 4.22, the overall proportion of verb phrases in functions and methods are 63.56%. The next most frequent is noun with

23.58%. Only about 5% of identifier names other than functions and methods are verb phrases. Its 4.99% for variables, 5.40% for parameters, 8.02% for attributes and 5.15% for classes respectively. The difference with functions/methods is obvious. We believe this indicates that programmers consciously choose verb phrases as identifier names when naming functions or methods.

Even though the data shows that verb phrases are consciously used as function/method names, there are still about 40% of the identifier names which belong to other grammatical structures. By reading and analysing these names in the source code, we found that there are 4 main reasons why they are not verb phrases. These are, grammatical errors, abbreviations, assigning multiple functions to a single function/method and accuracy of the classifier.

Firstly, grammatical errors in identifier names are very common but this does not mean that the programmers are not trying to use the verb phrases. We believe that there are maybe two reasons. The first is that the programmer's understanding of grammatical structure may be questionable. The second is that identifier names are different from standard prose. They are usually single words or phrases with multiple words and rarely appear as complete sentences. We believe this may lead to ignoring grammatical correctness while naming. One of the most common syntax errors we found is using adjectives to modify verbs, whereas the correct syntax is to use adverbs to modify verbs. For example, `random_flip` in `albumentations` and `safe_decode` in `borg`. The first method `random_flip` has the functionality to call the flip function in a package `openCV` which is used to flip a 2D array. The functionality of the second function `safe_decode` is to decode bytes to string when the status is safe. Although both decode and flip can be used as nouns, we believe that the use of each of these names is intended to describe the action indicated by its verb form. Their incorrect syntax causes them to appear as a noun. As mentioned in Section 3.2, the structure of function/method names should be grammatically correct. Therefore, the two identifier names in the example should be `randomly_flip` and `safely_decode`, which are using adverbs to modify verbs.

Secondly, the use of abbreviations can be separated into two parts which are abbreviated words and abbreviated phrases. Abbreviated words mean that some or all of the words are abbreviations. For example, `auth`, which can be found in 6 web-related repositories, is the abbreviation of `authorise`. The identifier name `cmp_assoc_values` in `sweetviz` is the abbreviation of `compare_associated_values`. Moreover, as mentioned in Section 4.1, there are 0.15% of the function/method identifier names are single letter. These abbreviated words may be classified as other incorrect parts of speech or be labeled as unknown part of speech ("X"). These cases would affect the results of whether they are verb phrases. In the above examples, the word `auth` is classified as a noun. The identifier name `cmp_assoc_values` is classified as a noun phrase, since the words are classified as

`X, X and Noun`, respectively. Obviously, both identifier names are verb phrases. Moreover, naming functions/methods with the abbreviation `auth` is common in web-related repositories.

Abbreviated phrases refer to the omission of a part of the name, especially the verb. For example, `to_string` in 8 different repositories and `max_width` in `flet`. By just reading the names themselves, they are not verb phrases and there are also no verbs involved. However, for both of the identifier names, their verb parts are removed. The functions/methods named `to_string` have the functionality to convert a variable of another type to a String type. The function `max_width` is used to get the maximum width value which is an attribute. Therefore, the full version of the two names in the example would be `convert_to_string` and `get_max_width`. The full versions are indeed verb phrases. We found many cases where the first verb word was omitted from the function/method name. Usually it is some verbs that are commonly used in functions/methods, such as `get` and `convert` mentioned in the example. We believe that in these cases, it does not affect comprehensibility very much. This is because the omission of these verbs does not affect the understanding of the functionality of functions/methods. Moreover, we think that these cases can also be thought of as consciously using the verb phrase to name functions/methods. Even if these phrases do not contain a verb, they still contain an action. Moreover, this kind of identifier names are widely used as listener functions/methods. The event listener functions and methods are typically used to record or modify the status associated with the event. Therefore, the identifier names are usually used to describe a completed action. For example, `on_mouse_press` in `manim` and `menu_open_clicked` in `AidLearning`. They describe the status of having clicked the mouse and the status of having clicked the button which can open the menu, respectively. However, the action described by these two identifier names is not actually the concept that the functions refer to. The verb `record`, which describes the functionality of these functions, has been omitted in both cases. We think that omitting the verb in these cases could be confusing.

The third reason for why the names are not verb phrases is related to another maintainability problem which is when the functions/methods try to do multiple things [39]. Then it becomes difficult to describe the concepts by using a phrase. This also makes it difficult for the identifier names to follow the naming conventions of using a verb structure and being grammatically correct.

The last reason is the accuracy of the classifier. As mentioned in Section 3.4.1, even with preprocessing, the accuracy of the classification of part of speech for identifier names is about 88%.

We investigated the use of verb phrases in different repository categories. Since using verb phrase is a naming convention related to functions and methods, we divided the data into these two parts as shown in Figure 4.23. In all the 6 repository categories, compared

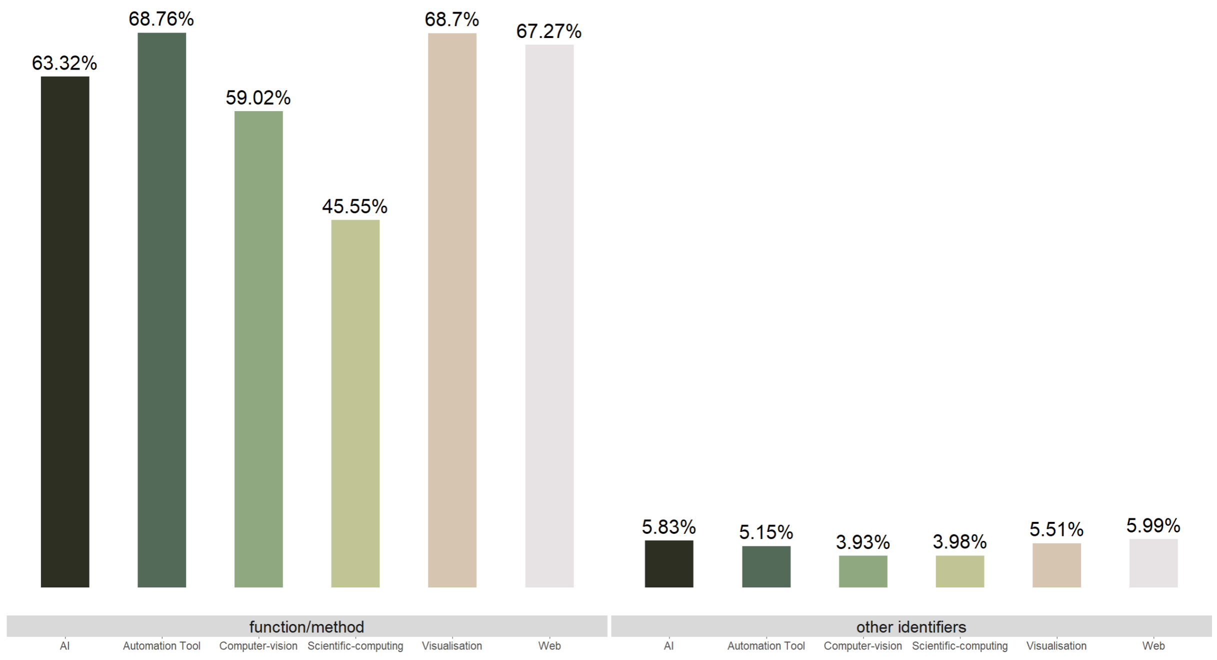


Figure 4.23: Verb phrase for function/method - Category

	LoC	Commits	Contributors	Releases	First Release
cor	0.0816	0.0658	0.0149	0.00253	-0.357
p-value	0.4	0.5	0.9	1	0.3

Table 4.4: Verb Phrase - Spearman correlation - function/method

to other kinds of identifier names, the data shows that programmers are consciously using verb phrases for naming functions and methods. We found that the probability of using verb phrase in repositories related to scientific-computing is lower compared to other categories. This might be related to that abbreviated phrases with omitted verbs are more common in scientific-computing. Other than that, by investigating the source code, we believe that categories do not have much relation with the use of verb phrases.

Since the difference between function/method names and others are large, we only investigated the correlation between the usage of verb phrases as function/method names and the properties of repositories. As shown in Table 4.4, the p-value of all the 5 properties are greater than 0.05. This indicates that for the data used in this thesis, there is no significant association involved.

We also investigated the use of verb phrases to name other kinds of identifier names which are not functions or methods. Firstly, we found that there is a positive correlation between the proportions of using verb phrases to name the identifiers (variables, parameters and attributes) in functions/methods and the scope length (Figure 4.24). Since the distribution matches the normal distribution, we used the Pearson correlation test. The p-value is 0.002 and the correlation is 0.553.

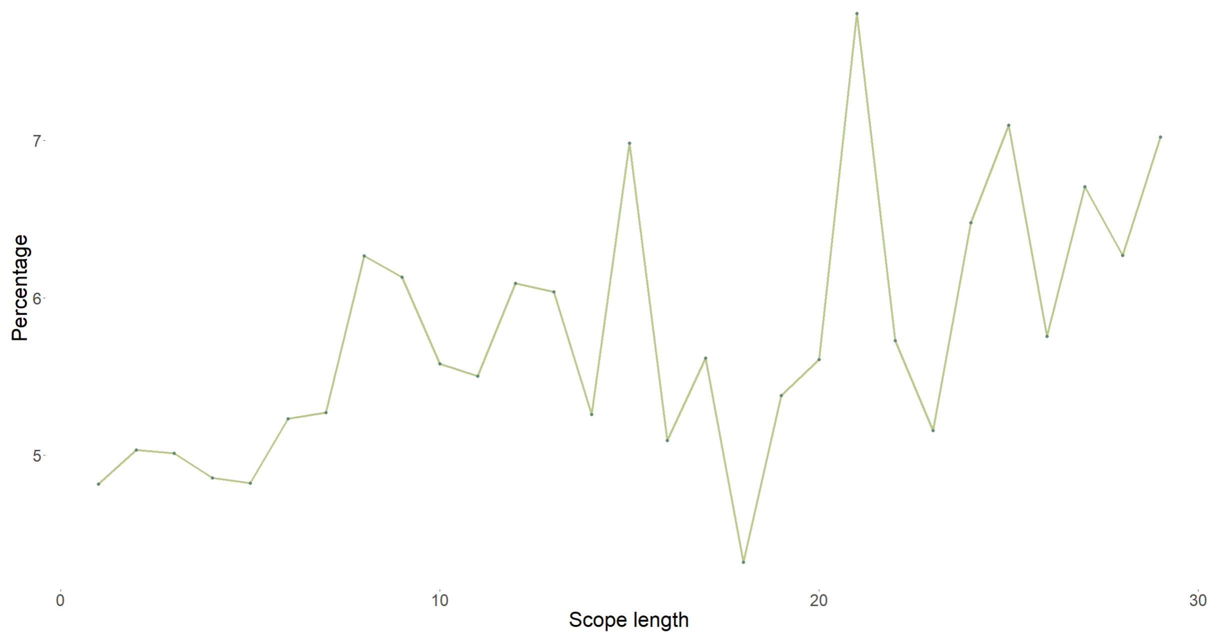


Figure 4.24: Verb Phrase - function/method scope length

We noticed that many of the verb phrases in other kinds of identifier names are related to inaccurate descriptions of the concepts. We found that in multiple repositories, verb phrases are used for assigning list or tuple types. For example, `create_table` is a variable that stores a tuple. We believe that this kind of identifier names are inaccurate for the describing the concepts. It should only keep the noun `table` and use some other words to modify it. This is because the process and action of creating the table is not important for a variable. When the variable is used again, the programmers do not need to know about the action of creating it and only care about which tuple it is.

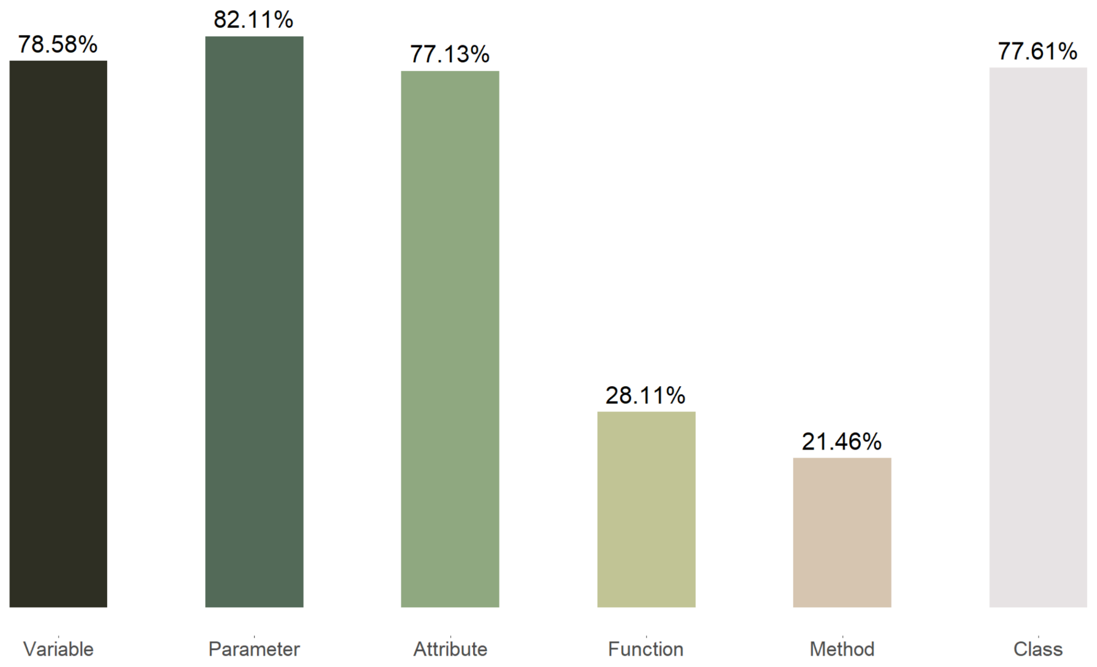


Figure 4.25: Noun phrase - Percentage

4.5 Noun Phrases for class names

Among all the 745,651 identifier names used in this thesis, the overall percentage of noun phrases is 71.20%. As shown in Figure 4.25, apart from functions and methods, most other kinds of identifier names are named by noun phrases. The proportions are 78.58% for variables, 82.11% for parameters, 77.13% for attributes, 28.11% for functions, 21.46% for methods and 77.6% for classes. Although this naming suggestion is related to class, variables, parameters and attributes also have similar proportions as classes. We think this is because the concepts of them all describe a specific object. There should not be any action involved in their concepts. Therefore, using noun phrases would be more cognitively consistent for identifier names except functions/methods. Since the part of speech related to functions/methods has already been explored in section 4.4, this chapter will not repeat it.

As shown in Figure 4.26, in most cases, programmers consciously choose to name classes using noun phrases. We investigated the cases where noun phrases are not used. We found that there are 2 reasons other than errors in the classifier that make the part of speech of class names to be of other kinds. As mentioned in Section 4.4, the accuracy of the classification of part of speech for identifier names is about 88%. The 2 reasons are abbreviations and wrong descriptions for concepts. Similar to the function/method names in Section 4.4, abbreviations in class names can be divided into abbreviated words and abbreviated phrases. The problem with abbreviated words is that it causes the classifier to fail to recognise the part of speech of the words. Some of these identifier

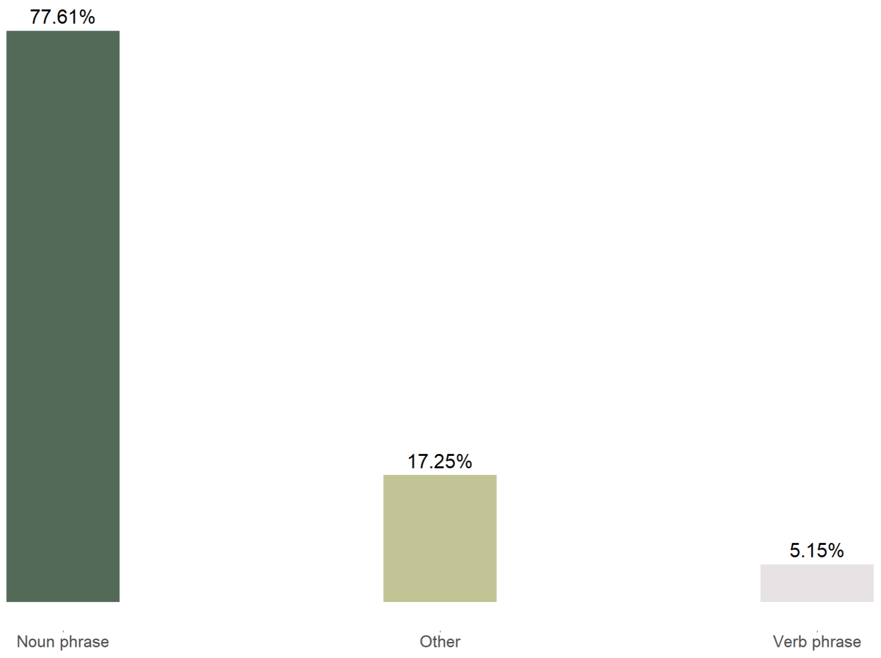


Figure 4.26: Class - Percentage

names are indeed noun phrases. For example, `ImgurIE` in `youtube-dl` is the abbreviation of `ImgurInfoExtractor`. In this case, information extractor is clearly a noun phrase. The word `imgur` is the name of a website which is used to modify `InfoExtractor`. We believe that these identifier names do follow the naming suggestion for using noun phrases as class names.

Regarding abbreviated phrases, we found that the practice in class names is to omit the noun and only keep the adjective. For example, `Lower` in `django` is a class related to lower case of letters. The full version of this class name should be `LowerCase`. We believe that such names have a negative impact on comprehensibility. Its different from the fact that most of the abbreviated phrases for function/method mentioned in section are common verbs. The omission in the Class name could be any of the nouns. Although it might be possible to figure out what is omitted by reading the comments and scope names, it is difficult to do so when programmers are only allowed to read the class names. Objects may be created and used in several different files, so we think they should keep the nouns in the class names. We think it is inappropriate to use a verb or verb phrase to describe a class name. This is because the concept of class does not contain any action. This also is applied to the other 3 kinds of identifier names (variables, parameters and attributes).

As shown in Figure 4.27 and 4.28, we investigated the use of noun phrases in different repository categories and found that there is no huge difference. The two reasons mentioned above appear in all 6 categories. We think it might be a consensus to use noun phrases for identifier names except functions/methods.

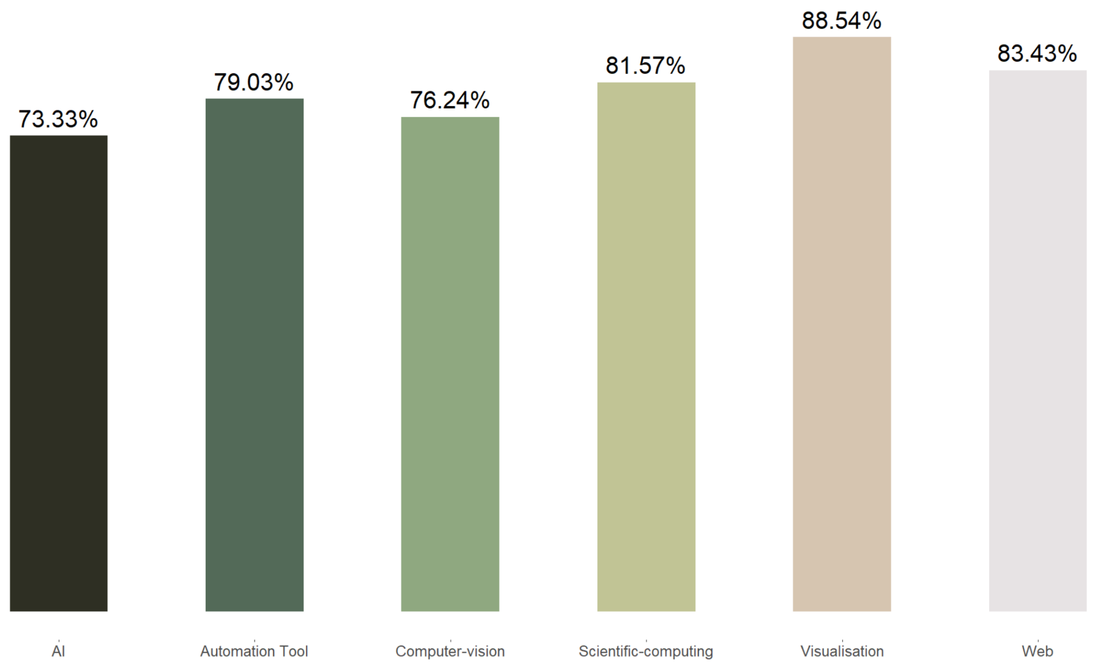


Figure 4.27: Noun phrase - Category - class

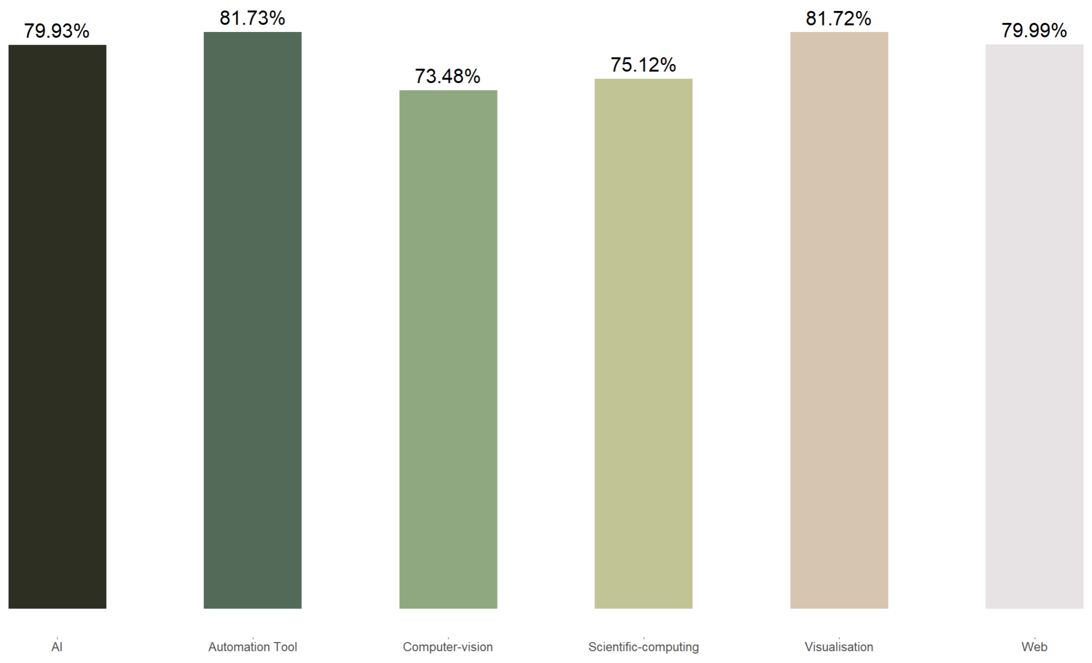


Figure 4.28: Noun phrase - Category - variable, parameter and attribute

	LoC	Commits	Contributors	Releases	First Release
cor	-0.33	-0.12	-0.092	-0.00715	0.07
p-value	0.001	0.3	0.4	1	0.553

Table 4.5: Noun phrase - Spearman correlation

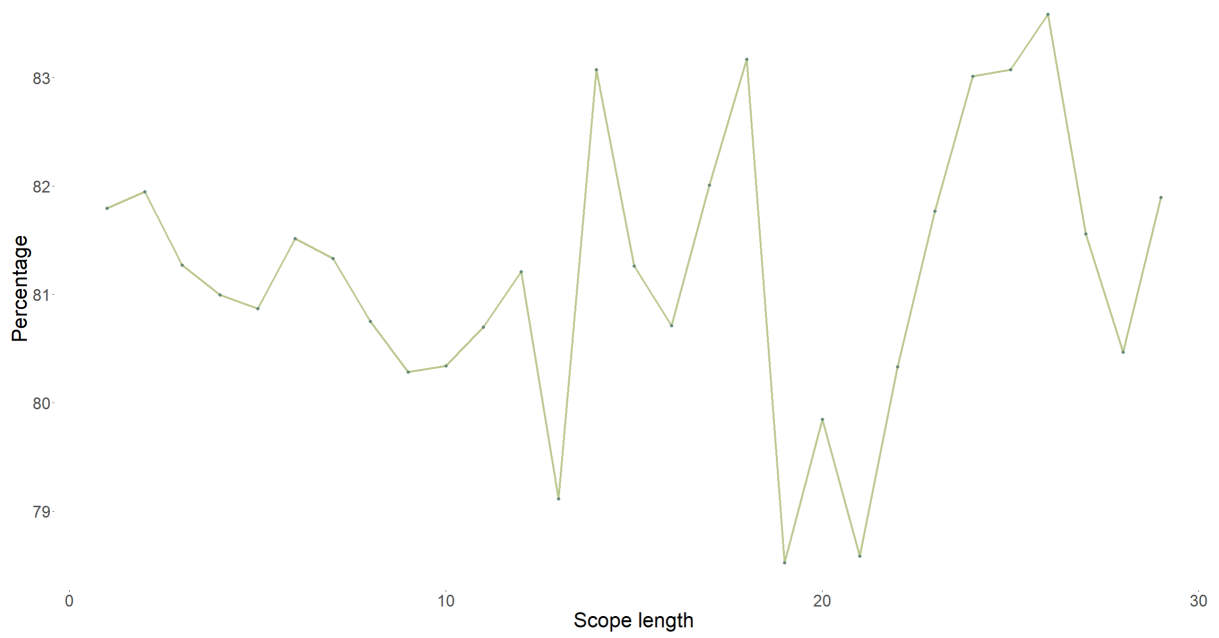


Figure 4.29: Noun Phrase - function/method scope length

We investigated the association between proportions and repository properties using noun phrases to name classes. Since the data is not normal distribution, we used the Spearman correlation test (Figure 4.5). The p-value of LoC is less than 0.05; however, the correlation is nearly zero. This means there is no correlation between LoC and using noun phrases as class names. There is no statistical evidence shows that any of the properties of repositories are related to the use of noun phrases for naming classes.

We found that noun phrases used in variables, parameters and attributes are similar to classes. The difference is that class names have more modifiers Which means there are more words in class names (Section 4.7). This is because classes usually contain larger and more complex concepts than the other 3 kinds of identifier names. Abbreviated phrases that omit the noun and retain only the adjective are also present in variables, parameters and attributes. Moreover, the use of noun phrases also has no relationship to context. The proportions are 88.97% for identifier names in loop statements and 81.88% for identifier names in If statements. We think this is reasonable since even in different contexts, the concepts that these identifiers refer to are all one or more objects. They should use noun or noun phrases to describe.

We also investigated the association between the scope length of functions/methods and the use of noun phrases (Figure 4.29). Since it fits to normal distribution, we used the Pearson correlation. The result is that p-value is 0.5 and correlation is 0.145. Therefore, in the data set used in this thesis, there is no significant association between using noun phrases to name variables/attributes/parameters and the scope length of the functions/methods.

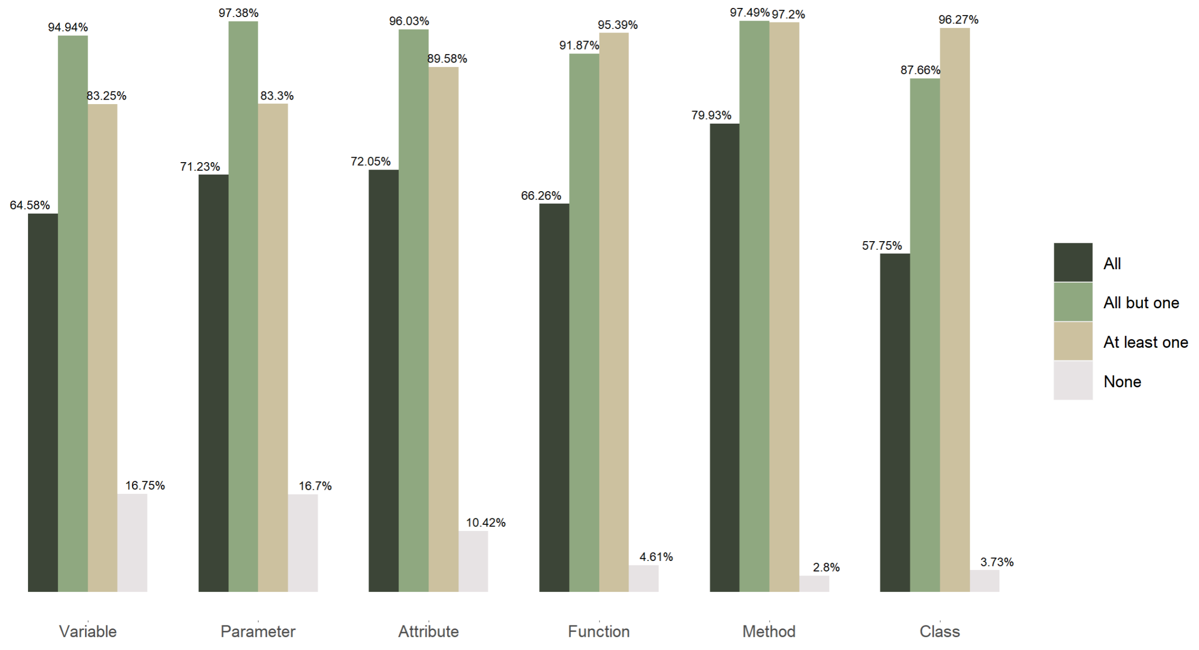


Figure 4.30: Dictionary word - Percentage

4.6 Dictionary words

As mentioned in Section 3.4.1, we investigated 4 types of data related to dictionary words, which are All, At least one, All but one and None (Figure 4.30). From all the 745,651 identifiers, we classified 514,486 (69.03%) names that all the words are dictionary words. 644,815 (86.52%) of them have at least one dictionary words. Among the remaining 13.48%, 4.26% are single letter identifier names. 713,279 (95.71%) of them contain at most one non-dictionary word. 100,462 (13.5%) of them does not contain dictionary words. As shown in Figure 4.30, the percentage for At least one and All but one are all above 80% across the 6 different kinds of identifier names. This indicates that most of the time, programmers are consciously using dictionary words to name identifiers. The proportion of both All and None in class names is relatively low. This might be because the concepts of classes are more complex. Programmers would use some abbreviations to reduce the length of the class names and also the time of reading these names.

We found that the reasons for not being categorised as dictionary words can be divided into 5 cases. These are, words that are commonly used in programming, abbreviations, accuracy of the classifier, identifier names that use numbers, and single letter names. The usage of last two has been discussed in Section 4.2, Section 4.3 and Section 4.1, so we are not going to repeat in this section.

Firstly, for the words are commonly used in programming, some of them are actually not dictionary words. For example, the word `Matlab` in `MatlabObject` and the word `backend` in `backend_name`. Neither of these words are dictionary words, but the use of them as names is not confusing. This is because they both have specific meanings and are not ambiguous. `Matlab` is a programming language which is widely used for scientific-computing and `backend` is the data access layer of a software. Since these kind of words are only used in the software domain, programmers should know the specific meaning of them. We believe that using these words does not cause any problems with comprehensibility and they are actually the same as dictionary words.

Secondly, abbreviations are commonly used in identifier names. These can be divided into three types, these are abbreviations that are commonly used in programming and abbreviations that have meaning in a specific domain and acronyms.

In addition to the previously mentioned words that are commonly used in programming, there are also abbreviations, such as the abbreviations `int` for `integer`, `http` for `Hypertext Transfer Protocol` and `gui` for `graphical user interface`. These kind of abbreviations would not cause any comprehensibility problems. This is because programmers would associate these abbreviations with the same concepts when they read the identifier names that contain these abbreviations.

Some of the abbreviations are meaningful in certain domains but this does not mean that they are all confusing. For example, the abbreviation `conv` is commonly used in identifier names related to convolutional neural networks. The problem with `conv` is that although programmers with a good understanding of the domain will probably think it is the abbreviation of `convolutional`, it can still be misinterpreted as the word `convention` or `convert`. These abbreviations, which have a high frequency of occurrence in a certain domain, should be consistent. That is, an abbreviation should only refer to one concept in a repository. If they are not used very often, these identifier names should use dictionary words. Some other common abbreviations, such as `num` stands for `number`, are not considered to have an impact on comprehensibility. Since they are widely used, they would not lead to multiple concepts which is not confusing. However, the other abbreviation for number `no` can be confusing. This is because this abbreviation should be `no.`, but it is not allowed to have `.` in identifier names.

The acronyms are not like the previously mentioned abbreviations. They are usually not associated with a particular domain. The class name `RCR` and `RRR` in `AITemplate` are two exmaples of the initials of multiple words. The letter `R` refers to `Row` and `C` refers to the `Column`. The problem of these two names is obvious. It is impossible to understand the meanings of names before reading the comments. We think it is better to use the more obvious names `RowColumnRow` or `RowColRow` in this case. The `Col` in the name `RowColRow` can also be deduced to refer to the word `Column` when only reading the identifier name.

Moreover, the classifier error also needs to be taken into account. This is because of the unavoidable problem of using spell checker to detect dictionary words. If an abbreviation exists in the dictionary with the same spelling but a completely unrelated meaning, it

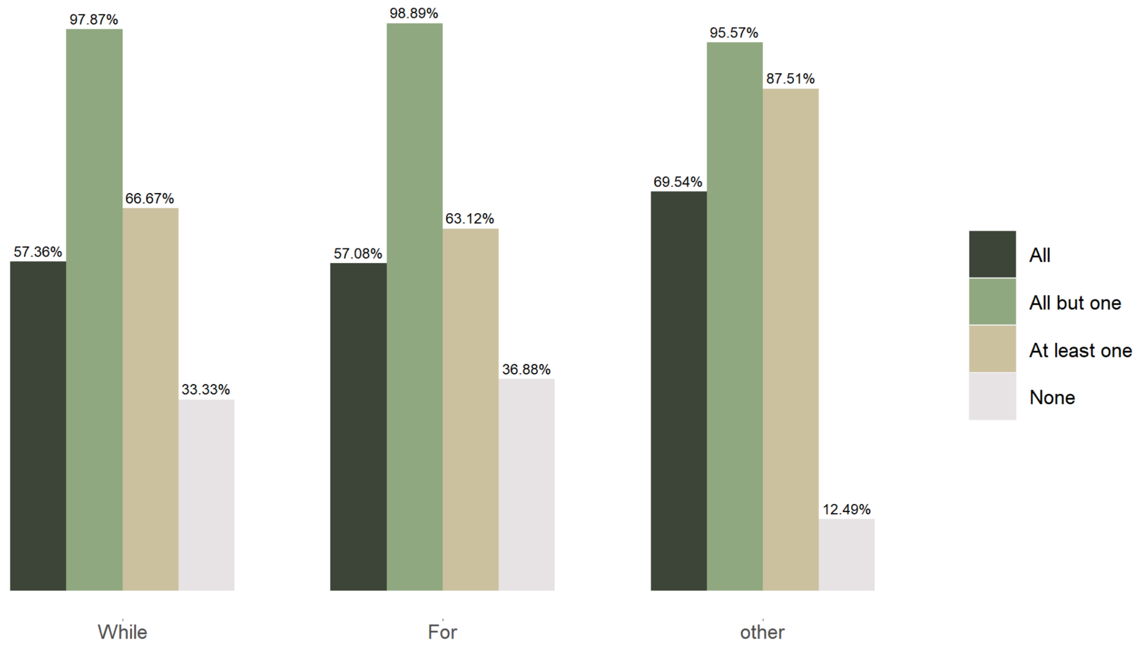


Figure 4.31: Dictionary word - Loop

would still be labelled as a dictionary word by the classifier. For example, the word `del` in `del_statement` from `moto` is the abbreviation of delete. However, the word `del` is also the word for a mathematical operator. This case is not very common and most words are labelled correctly.

We investigated the use of dictionary words in class names. The percentage of using all dictionary words is the lowest which is 57.75% which is the only kind of identifier name with proportion less than 60%. Moreover, the proportion of class names that contain at least one dictionary word is less than that have at most one non-dictionary word. This is unique among the 6 kinds of identifier names. This means that it is more common to use multiple abbreviations in class names than in the other kinds. This might be related to that the classes are more likely referring to more complex concepts than other kinds of identifiers. Then they require more words to modify the nouns. To shorten the length of the name, programmers would choose to use abbreviations. However, as mentioned in the paragraph about abbreviations above, we believe that directly reducing the length of names in this way would cause negative effect to comprehensibility.

In variables, parameters and attributes, the use of all dictionary words for naming variables is relatively low (Figure 4.30). The proportions for the other two types of data are similar to parameters and attributes. We investigated the variables in different contexts. For the variables in If statements, they are more likely to be full dictionary words (74.33%). The proportions are relatively low in both loop statements which are 57.06% for variables in For loop statements and 57.36% for variables in While loop statements (Figure 4.31). Moreover, as shown in Figure 4.32, the use of dictionary words are very

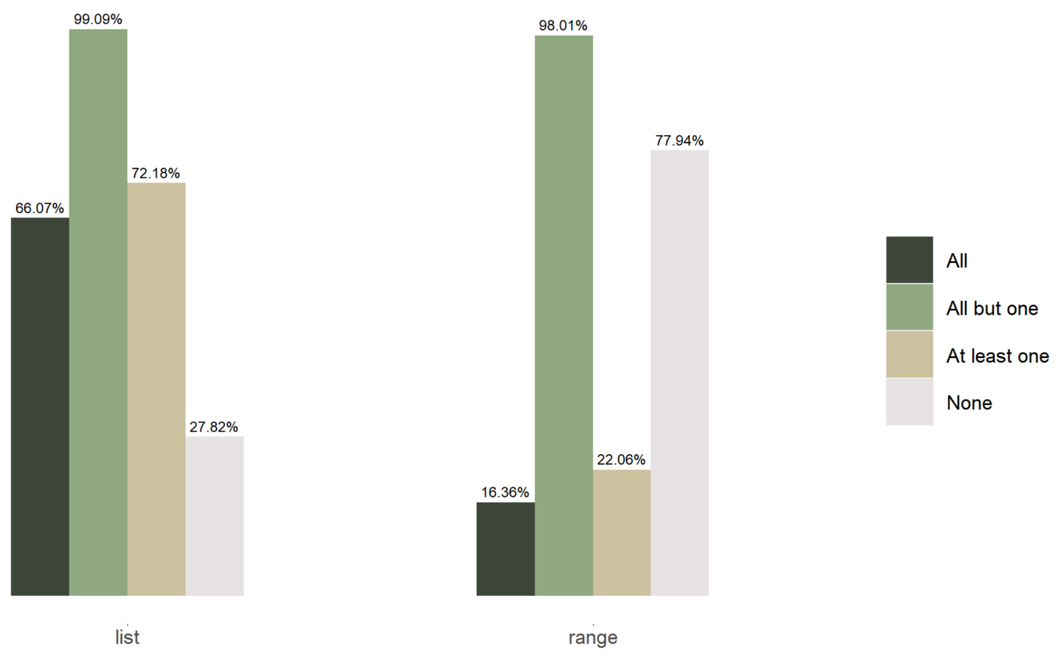


Figure 4.32: Dictionary word - For

different in list type and range type For loop statements. For variables used in range type statements, the proportions of all dictionary words (16.36%) and at least one dictionary words (22.06%) are low, but almost all identifier names contain at most one non-dictionary word (98.01%). This is because single letter names are commonly used in the range type statements which has been discussed in Section 4.1.

We also investigated the relevance of scope length for the use of dictionary words in variables, parameters and attributes. Since the data does not have normal distribution, we use the spearman correlation. The result is that p-value is 0.9 and correlation is -0.0135. This indicates that there is no correlation involved. The use of dictionary words is not affected by the scope length.

As shown in Figure 4.33, the use of dictionary words are different in various repo categories. The proportions of dictionary words used in computer-vision and scientific-computing are relatively low. As mentioned in Section 4.1, Section 4.2 and Section 4.3, this is related to the greater use of single letters and numbers for identifier names in these repositories.

By the Spearman correlation, we investigated the correlation between proportions of dictionary words used and proportions of the repositories. As shown in Table 4.6, the `All but one` has association with LoC. The correlation of LoC is -0.475 which means smaller repositories have higher proportions. There is no evidence showing a correlation between other proportions and the use of dictionary words.

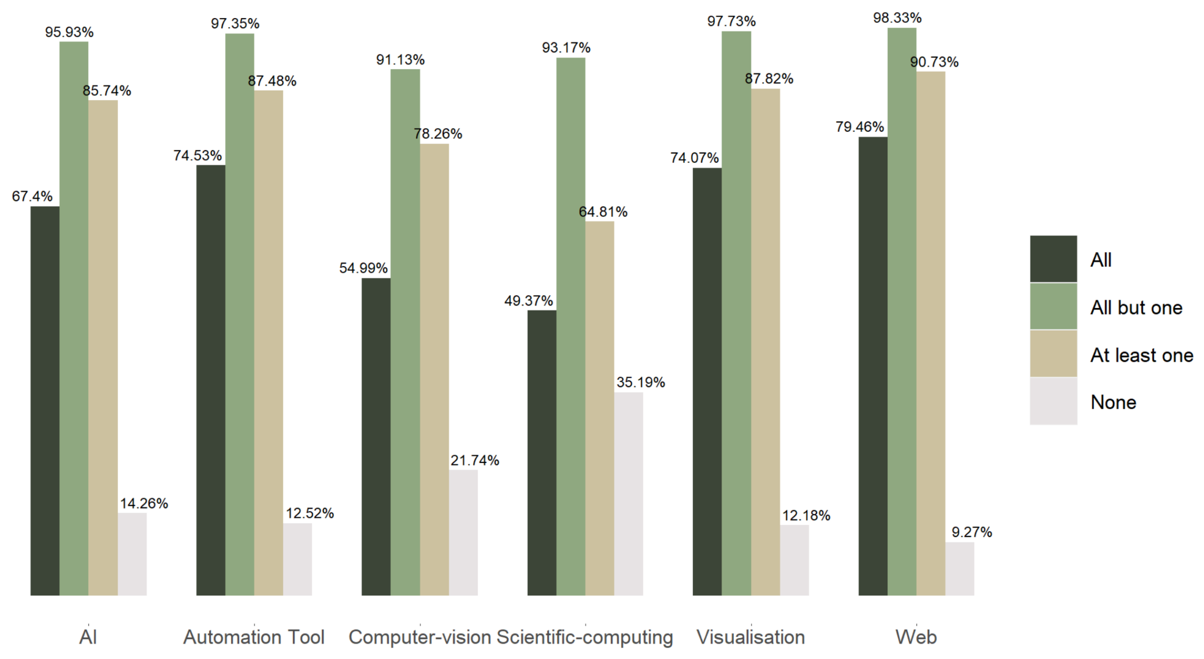


Figure 4.33: Dictionary words - Category

		LoC	Commits	Contributors	Releases	First Release
All	cor	0.0309	0.162	0.0876	0.137	0.189
	p-value	0.8	0.1	0.5	0.3	0.6
At least one	cor	0.187	0.176	0.111	0.139	-0.109
	p-value	0.06	0.08	0.3	0.3	0.7
All but one	cor	-0.457	-0.176	-0.0533	0.124	0.545
	p-value	$2x10^{-6}$	0.08	0.6	0.4	0.07
None	cor	0.06	0.08	0.3	0.3	0.6
	p-value	-0.187	-0.176	-0.111	-0.139	0.154

Table 4.6: Dictionary words - Spearman correlation

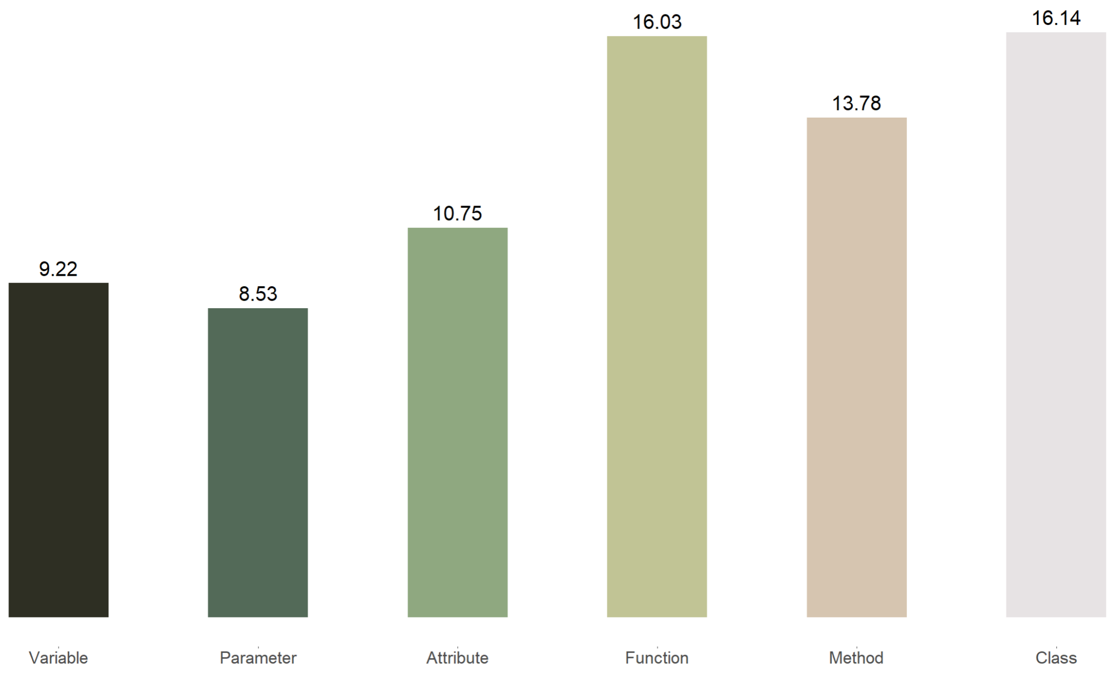


Figure 4.34: Length - Average

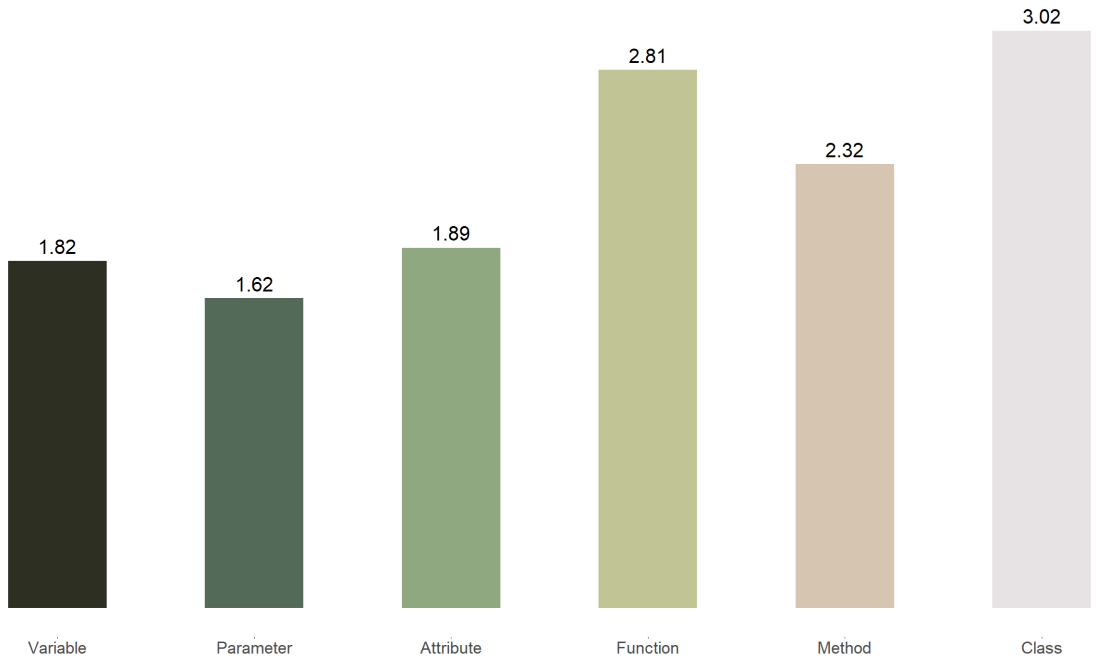


Figure 4.35: Number of words - Average

4.7 Length and number of words

In this section, we will discuss the length of identifier names from two ways of measuring which are the number of characters (length) and the number of words. Among the 745,651 identifiers names investigated, their average length is 10.20. Figure 4.34 shows that the average length are different in different kinds of identifier names. Variables (9.22), param-

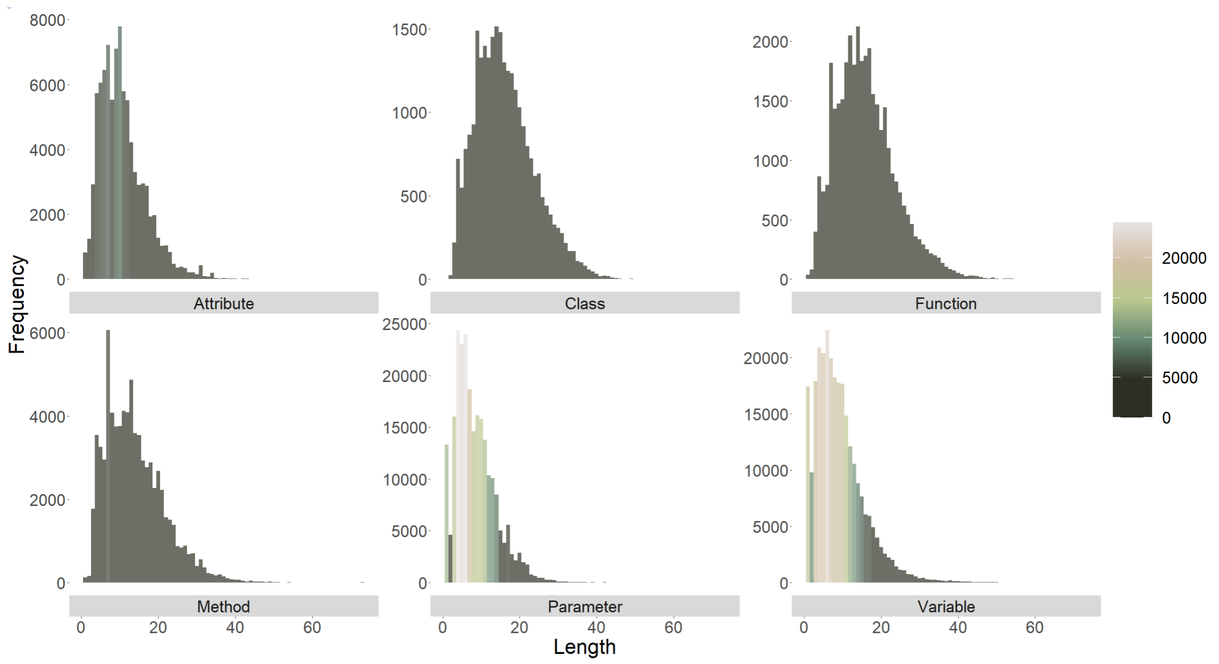


Figure 4.36: Length - Frequency

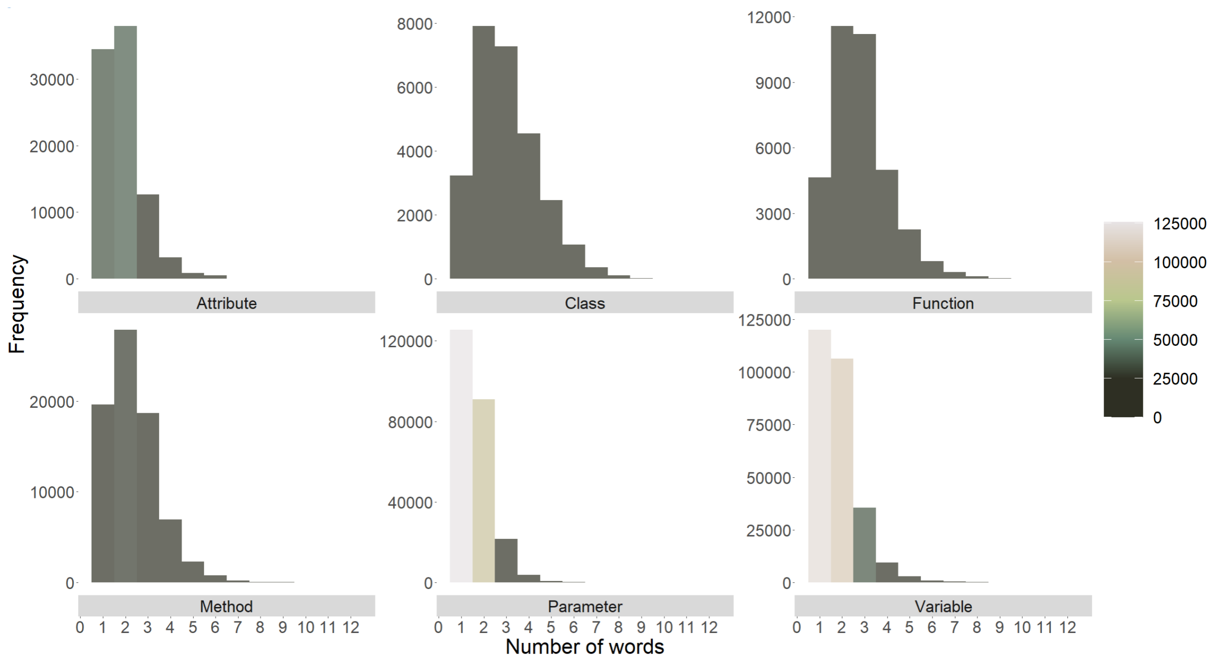


Figure 4.37: Number of words - Frequency

eters (8.53) and attributes (10.75) are shorter. The other three kinds are longer which are 16.03 for functions, 13.78 for methods and 16.14 for class names. As shown in Figure 4.35, the average number of words used in identifier names have a similar trend to the average length of the name. They are 1.82 words for variables, 1.62 words for parameter, 1.89 words for attributes, 2.81 for functions, 2.32 for methods and 3.02 for classes.

The longest identifiers among the 100 repositories is the function name `_generate_dummy_inputs_for_sequence_classification_and_question_answering` in `FlexGen`. It uses 12 words and the length is 73. As shown in Figure 4.36 and Figure 4.37, the trend of length and number of words varies across kinds of identifier names. The most common name lengths are 6 for variables, 4 for parameters, 10 for attributes, 14 for functions, 7 for methods and 14 for classes. Variables and parameters are most commonly named with 1 word, while the others are 2 words.

The average number of characters (length) of each word are 5.07 for variables, 5.27 for parameters, 5.69 for attributes, 5.70 for functions, 5.94 for methods, 5.34 for classes. These are generally consistent with the results in Section 4.6. Since functions and methods have the highest percentage of all dictionary words used as identifier names, each word is longer. Generally full words are longer than abbreviations. The identifier names with more dictionary words should be longer than non_words. For example, the number in middle name `str2bool` is shorter than `strToBool`. Although variables and parameters have a higher percentage of dictionary words, they have higher average number of characters than class names. This might be related to the classification error. This is because acronyms such as `RCR` and `RRR` is treated as a word by `SpaCy`, but they are actually 3 words. As mentioned in Section 4.6, acronyms are used more often in class names. Moreover, Section 4.1 mentioned that variables and parameters have higher proportion of single letter identifier names than class names which means the error has less effect on them. This might have an impact on the results.

The class names are the longest and the average number of words used is the only one greater than 3. This is surprising due to the error mentioned in last paragraph and also class names have the lowest usage of dictionary words (Section 4.6). Even in cases where more abbreviations and numbers are used in classes names, longer names are still needed to describe the classes. This means that classes contain more complex concepts than other kinds of identifiers.

By using Spearman correlation, the length of variables, parameters, and attributes used in functions and methods, and also number of words are correlated with the scope length of functions/methods. The results for length are that p-value is $3x10^{-4}$ and correlation is 0.622. Moreover, the results for number of words are that p-value is $2x10^{-5}$ and correlation is 0.709. These results show a trend that longer identifier names are more likely to be used in larger functions. This might be because the concept of identifiers is relatively more complex in large scopes, it is more difficult to find where identifiers were first created and also there are more identifiers in total. This means that longer names need to be used to accurately describe these identifiers.

The identifiers names used in the 4 types of statements investigated in this thesis are all shorter compared to the others (Figure 4.38). 80% of these identifier names used less than 3 words, and less than 12 characters (length). This is most obvious in the

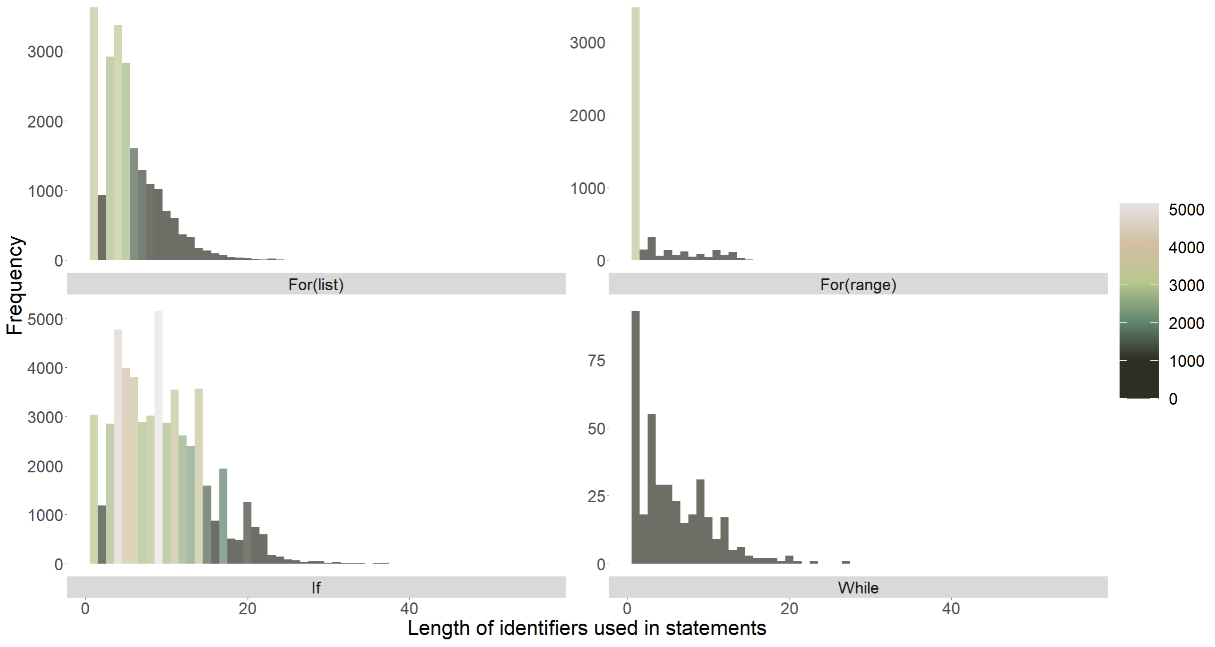


Figure 4.38: Length - Frequency - Statements

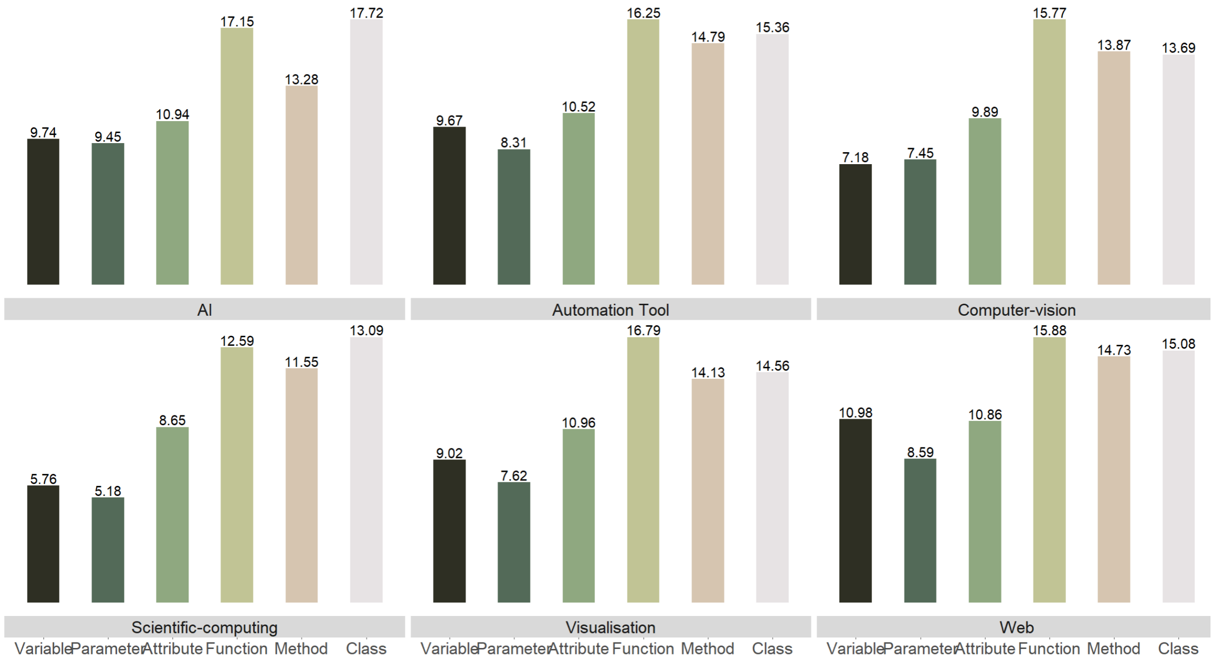


Figure 4.39: Length - Category

range type for statements. 84% of them use only one word. This might be because the simpler concept of identifiers in statements. As well as these identifier names are usually temporary, so it is more convenient to use shorter names.

As shown in Figure 4.39 and Figure 4.40, we investigated that the length and number of words used for identifier names varies across repo categories. The identifiers in scientific-computing and computer-vision are relatively short. This matches the results in single

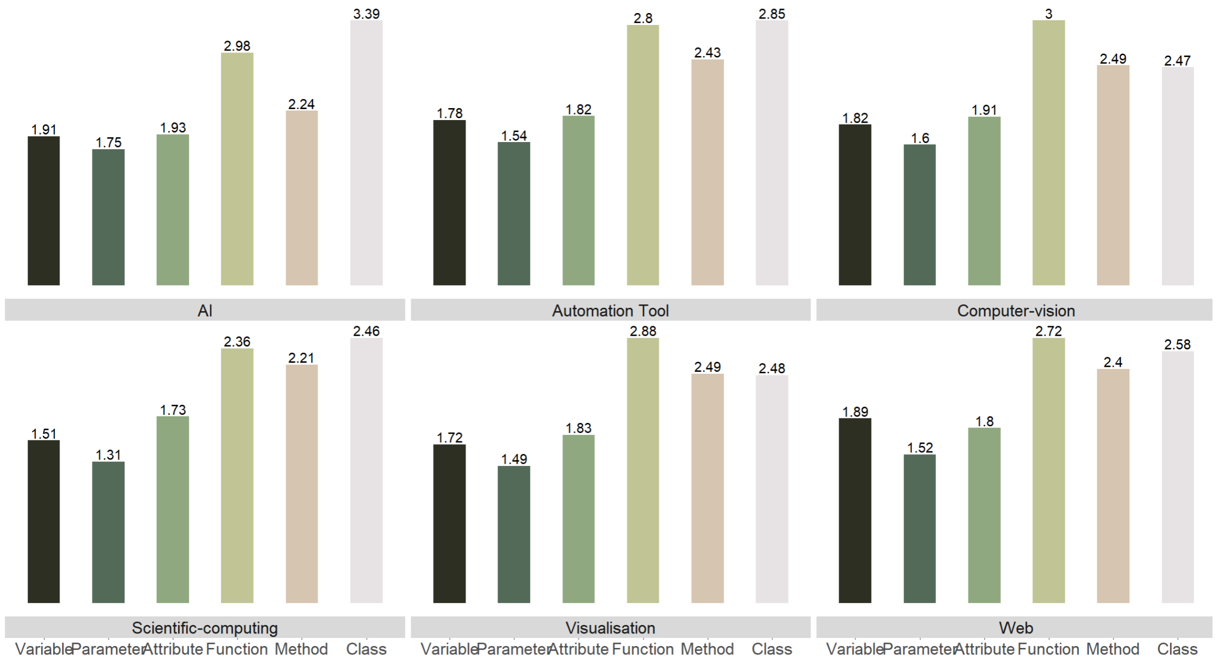


Figure 4.40: Number of words - Category

		LoC	Commits	Contributors	Releases	First Release
Length	cor	0.278	0.117	0.0377	0.0748	-0.336
	p-value	0.005	0.2	0.7	0.6	0.3
number of words	cor	0.36	0.107	-0.0177	0.0475	-0.79
	p-value	3x10^{-4}	0.3	0.9	0.7	0.004

Table 4.7: Length and Number of words - Spearman correlation

letter, number at end, number in middle, and dictionary words. When class names are excluded, the names in automation tool have more words than in computer-vision, but the average length is shorter. This means that the identifier names in the automation tool usually consist of longer words.

We investigated the association between length/number of words and repository properties. As shown in Table 4.7, since the data does not have normal distribution, Spearman correlation is used here. Firstly, the results show that both length and number of words are correlated with LoC. Larger repositories tend to use longer identifier names. Figure 4.41 shows that this trend exists in all the 6 kinds of identifier names. Secondly the data also shows that first release is correlated with number of words, but it is not correlated with name length. The correlation with number of words is -0.79. This indicates that identifier names in repositories created earlier used more words, but their name length are not longer. The might because they use more abbreviations. The results also show that the other 3 repository properties are not associated with neither length or number of words.

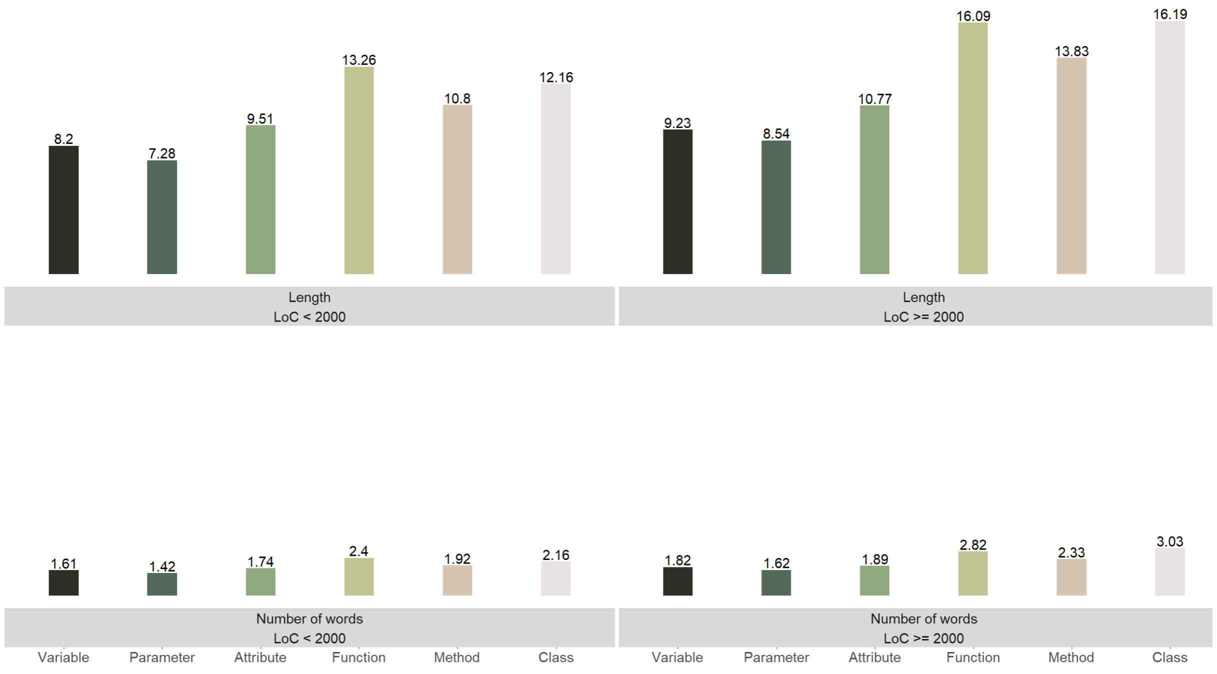


Figure 4.41: Length and Number of words - LoC

	Identifier Names	Number of Repetitions	Number of Repositories	Number of Categories	Variable %	Parameter %	Attribute %	Function %	Method %	Class %
1	name	5286	81	6	23.33	66.1	8.65	0	1.93	0
2	x	4273	64	6	21.93	76.34	1.15	0.02	0.56	0
3	value	4165	75	6	17.17	75.97	5.62	0.05	1.2	0
4	config	3891	53	6	13.62	84.91	1	0.31	0.13	0.03
5	data	3487	80	6	41.07	50.24	7.69	0.09	0.92	0
6	i	3085	77	6	91.8	7.46	0.71	0	0.03	0
7	forward	2852	20	6	0.07	0.21	0.18	0.35	99.19	0
8	key	2706	75	6	45.97	47.04	6.36	0.15	0.48	0
9	args	2121	74	6	34.28	60.68	3.91	0	1.13	0
10	attention_mask	1945	5	1	3.7	95.84	0.46	0	0	0
11	url	1875	54	6	19.79	76.11	1.71	0.16	2.24	0
12	output_attentions	1857	2	1	0.05	97.79	2.15	0	0	0
13	y	1820	50	6	26.48	70.22	1.98	0	1.32	0
14	logger	1688	50	6	94.25	4.21	1.24	0.06	0.24	0
15	n	1681	54	6	48.9	48.3	2.74	0	0.06	0
16	model	1674	39	6	27.72	55.62	15.53	0.18	0.96	0
17	output	1647	51	6	50.46	30.12	18.52	0.12	0.79	0
18	d	1631	60	6	81.61	16.25	1.96	0.06	0.12	0
19	path	1631	70	6	30.41	65.11	2.51	0.06	1.9	0
20	hidden_states	1600	3	1	2.25	92.25	5.5	0	0	0
21	result	1556	71	6	87.6	7.13	3.86	0.32	1.09	0
22	a	1469	52	6	31.45	67.39	1.09	0	0.07	0
23	k	1453	59	6	66.14	31.31	2.55	0	0	0
24	_backends	1433	3	3	0.07	0	99.93	0	0	0
25	state	1429	44	6	14.77	71.8	12.67	0	0.77	0
26	params	1417	53	6	42.13	51.66	5.72	0	0.49	0
27	dtype	1386	31	6	17.6	75.47	4.26	0.07	2.6	0
28	input_ids	1377	4	1	8.57	91.29	0.15	0	0	0
29	output_hidden_states	1359	2	1	0	97.87	2.13	0	0	0
30	return_dict	1333	4	1	0	98.65	1.35	0	0	0
31	X	1319	22	5	5.08	94.54	0.3	0	0.08	0
32	item	1278	62	6	66.9	32.24	0.86	0	0	0
33	parser	1259	53	6	76.89	22.24	0.32	0.08	0.48	0
34	obj	1236	54	6	25.57	71.68	2.67	0	0.08	0
35	other	1223	47	6	1.72	97.87	0.41	0	0	0
36	filters	1201	14	5	21.9	76.85	1.08	0	0.17	0
37	size	1198	53	6	25.38	60.43	9.27	0.25	4.59	0.08
38	head_mask	1164	2	1	0.6	99.4	0	0	0	0
39	func_attrs	1154	1	1	0.09	99.91	0	0	0	0
40	dropout	1117	13	4	0.63	17.28	80.93	0.45	0.72	0
41	index	1094	60	6	38.57	45.52	14.08	0.64	1.19	0
42	p	1089	59	6	53.17	45.36	1.29	0.09	0.09	0
43	__all__	1084	45	6	100	0	0	0	0	0
44	inputs	1074	26	6	16.48	64.34	13.04	0	6.15	0
45	s	1068	65	6	56.74	37.45	5.71	0	0.09	0
46	df	1066	25	6	71.58	26.17	1.59	0.66	0	0
47	func	1059	58	6	28.23	64.21	4.25	2.74	0.57	0
48	_VALID_URL	1048	1	1	0	0	100	0	0	0
49	template	1034	34	6	79.4	11.9	7.74	0.19	0.77	0
50	request	1022	31	4	5.58	92.86	0.98	0.29	0.29	0

Table 4.8: Most common identifier names

4.8 Most common names

As shown in Table 4.8, we investigated the 50 most common identifier names among all the 745,651 names. They used a total of 89,615 times which are 11.98% of all identifier names. The three most common names are `name`, `value` and `x` that all used more than 4000 times. These names are almost always used for identifiers other than class names, because in python naming convention, class names usually use CamelCase with leading upper case letters. Only `config` and `size` are used for class names, which we believe do not follow the naming convention. They should use `Config` and `Size` to name the corresponding classes. 40 of these names contain only 1 word, 9 names have 2 words and only 1 name has 3 words.

There are 10 identifier names that are single letters, which are `x, i, y, n, d, a, k, X, p and s`. As mentioned in Section 4.1, over 90% of these single letters are used to name variables, attributes and parameters. 9 of these are lowercase. Both uppercase and lowercase x are commonly used identifier names. These usages have been discussed in Section 4.1, so we are not going to repeat in this section.

Except for the 9 single letters mentioned above, 11 of the top 50 identifier names are or contain abbreviations. Moreover, 2 of them are acronyms which are `url` and `df`. The identifier name `url` is commonly used in programming which is the abbreviation for `Uniform Resource Locator`. This would not lead to confusion. For the identifier name `df`, it is being used in 25 different repositories and all the 6 repository categories. We found that it refers to different concepts in different repositories. For example, it is an abbreviation for `data frame` in `catalyst`, and `dominance frontiers` in `angr`. These usages can lead to misunderstandings especially in such very common identifier names. They should use the full words to name these identifiers.

Nine of these names are only used in less than or equal to 5 different repositories. Most of them belong to only 1 repo category. This means that they are commonly used words in a particular field. For example, the identifier name `attention mask` can be found in 5 AI-related repositories. The attention mask is a binary tensor used to prevent the model from paying attention to the padded indices by indicating the position. Therefore, It is a word that is only used for training models, so it would only be used in AI-related repositories. Of all the 9 identifier names that appear in only 1 repo category, 8 are AI-related. Only `_VALID_URL` is found only in automation tool related repositories. There are two possible reasons for this. Firstly, identifier names in AI-related repositories are the most numerous that is 367,805. Secondly, there are more commonly used proper names in AI-related fields compared to other categories.

Since variables, parameters and attributes make up the majority of identifiers, we also investigated the 50 most common names in functions/methods and classes separately. As shown in Table 4.9 and Table 4.10, using the same names for different identifiers also exists in both functions/methods and classes. Their number of repositories and number of categories are relatively less. This is particularly evident in the class names. Only one of the class names (`Model`) appears in all 6 repo categories and only `Config` is used in more than 10 different repositories. This might be because functions, methods and classes contain more complex concepts, so there are fewer cases to use the same identifier names. The top 50 function/method names have a total of 13925, which are 12.36% of all function/method names. The top 50 class names have a total of 1,235, which are 4.57% of all class names.

	Identifier Names	Number of Repetitions	Number of Repositories	Number of Categories
1	forward	2839	17	6
2	call	974	13	5
3	_real_extract	968	1	1
4	execute	645	17	5
5	run	538	47	6
6	setup	361	15	5
7	validate	350	19	6
8	custom_forward	280	2	1
9	create_custom_forward	279	2	1
10	get_test_params	266	1	1
11	get_config	251	12	5
12	copy	250	20	6
13	get_input_embeddings	241	2	1
14	update	240	43	6
15	serving_output	224	1	1
16	get	219	40	6
17	gen_function	212	1	1
18	build	204	19	5
19	_fit	198	2	2
20	check	198	16	6
21	set_input_embeddings	194	2	1
22	get_data	189	17	5
23	main	180	54	6
24	fit	175	13	5
25	step	173	12	6
26	to_dict	163	15	5
27	get_output_embeddings	158	2	1
28	render	157	18	5
29	reset	156	30	6
30	set_output_embeddings	151	2	1
31	apply	142	18	4
32	load	142	34	6
33	process	142	17	6
34	_init_weights	140	4	2
35	_transform	140	2	2
36	get_columns	137	5	5
37	close	135	35	6
38	gen_function_call	132	1	1
39	gen_function_decl	129	1	1
40	predict	129	13	5
41	prepare_inputs_for_generation	125	2	1
42	to_json	124	10	4
43	on_update	119	3	2
44	_predict	109	3	2
45	_prune_heads	109	2	1
46	delete	109	20	5
47	save	109	26	6
48	_set_gradient_checkpointing	107	2	1
49	add	107	30	6
50	backward	106	8	3

Table 4.9: Most common identifier names - Function/Method

The identifier names commonly used to name functions/methods and classes are relatively longer but do not exceed 4 words. Of the 50 names for functions and methods, 28 contain 1 word, 12 contain 2 words, 9 contain 3 words, and 1 contains 4 words. Similarly, for classes, 30 identifier names contain 1 word, 13 contain 2 words, 5 contain 3 words and 2 contain 4 words.

The commonly used `get` and `set` methods are also present in top 50 identifiers. There are 7 of them related to `get`. This also includes the single word name `get`. We think it

	Identifier Names	Number of Repetitions	Number of Repositories	Number of Categories
1	Meta	333	9	4
2	Migration	271	4	3
3	Config	108	15	5
4	Command	56	7	4
5	ResourceNotFoundException	32	2	3
6	BasicTokenizer	20	2	1
7	WordpieceTokenizer	20	2	1
8	ValidationException	19	1	2
9	BenchmarkLayer	16	1	1
10	E2E	15	1	1
11	Encoder	15	4	1
12	Model	15	9	6
13	Node	14	10	4
14	Host	12	3	3
15	MyApp	11	1	1
16	ValidationError	11	4	3
17	Arbiter	10	1	1
18	Guest	10	1	1
19	InvalidParameterException	10	1	2
20	LayerNorm	10	4	1
21	Table	10	5	4
22	DatabaseWrapper	9	1	1
23	InvalidRequestException	9	1	2
24	Serializer	9	2	3
25	StorageSession	9	1	1
26	StorageTable	9	1	1
27	Block	8	6	4
28	CustomConverter	8	1	1
29	DatabaseFeatures	8	1	1
30	Media	8	3	3
31	ResourceNotFound	8	2	2
32	Session	8	7	3
33	Transformer	8	4	1
34	User	8	5	2
35	BadRequestException	7	2	3
36	BiasLayer	7	1	1
37	Decoder	7	2	1
38	Identity	7	5	2
39	Index	7	5	2
40	Layer	7	4	4
41	Operation	7	7	4
42	Pipeline	7	6	4
43	ResourceInUseException	7	1	2
44	Sequential	7	5	2
45	State	7	6	5
46	Task	7	6	3
47	Attention	6	3	1
48	BaseObject	6	3	5
49	Bottleneck	6	4	2
50	Client	6	4	3

Table 4.10: Most common identifier names - Class

is difficult to understand what it is getting when only reading the identifier name itself. Therefore, the noun should be added to describe the objects to be fetched.

	Words	Number of Repetitions	Number of Repositories	Number of Categories	Variable %	Parameter %	Attribute %	Function %	Method %	Class %
1	name	20531	92	6	31.2	48.56	13.26	1.78	4.72	0.48
2	get	16848	93	6	2.15	0.59	0.77	34.89	60.92	0.66
3	data	12349	88	6	37.34	33.4	9.58	6.86	8.98	3.84
4	id	11941	71	6	39.73	41.59	9.65	2.22	6.11	0.7
5	type	11127	83	6	29.61	42.18	13.37	3.98	7.94	2.92
6	output	10610	70	6	29.33	46.23	9.35	1.15	8.76	5.18
7	x	9688	70	6	35.17	56.16	4.1	0.84	1.84	1.9
8	model	9021	47	6	27.42	25.98	17.44	3.76	8.24	17.16
9	size	8727	77	6	31.06	43.66	17.97	1.57	5.23	0.52
10	mask	8720	41	6	23.67	64.16	4.76	2.29	4.22	0.89
11	value	8574	78	6	29.94	50.07	7.57	2.44	8.13	1.85
12	key	8292	86	6	42.08	36.93	7.51	4	7.36	2.12
13	config	8188	67	6	24.17	49.16	8.84	4.38	6.4	7.05
14	to	8097	86	6	22.5	17.66	12.23	17.12	28.22	2.27
15	input	7398	66	6	28.82	48.61	9.77	1.92	9.98	0.91
16	path	7083	90	6	36.28	42.79	10.53	4.42	4.6	1.37
17	list	6571	85	6	43.92	19.95	8.99	6	17.97	3.17
18	url	6228	71	6	34.65	32.84	23.39	2.68	6.02	0.42
19	2	6171	75	6	48.06	14.39	10.79	11	2.53	13.22
20	ids	6013	50	6	25.03	66.17	2.51	1.41	4.72	0.15
21	d	5996	70	6	46.05	30.77	6.5	8.76	2.99	4.94
22	num	5830	69	6	35.33	36.71	22.18	1.84	3.74	0.21
23	attention	5694	13	3	18.41	55.97	13.58	0.77	3.44	7.83
24	n	5432	65	6	50.24	37.56	8.41	1.31	1.47	1.01
25	token	5413	43	6	25.24	52.37	7.59	1.37	8.76	4.67
26	layer	5362	29	6	34.19	20.78	27.14	2.35	2.65	12.91
27	is	5306	86	6	24.76	16.74	18.77	18.21	20.77	0.75
28	hidden	5111	30	6	11.74	76.83	9.47	0.12	1.64	0.2
29	1	5074	69	6	59.7	21.17	11.43	4.26	1.04	2.4
30	dict	5048	75	6	30.27	47.5	5.25	5.23	10.3	1.45
31	max	4952	79	6	35.7	38.97	16.11	2.99	4.42	1.8
32	set	4861	76	6	11.97	11.71	5.88	9.67	56.26	4.51
33	shape	4790	42	6	48.27	31.57	8.68	2.94	7.89	0.65
34	state	4782	60	6	20.51	51.17	10.75	2.4	12.57	2.59
35	doc	4750	43	6	62.19	23.64	6.21	5.26	2.15	0.55
36	states	4664	30	6	16.47	75.11	6.33	0.36	1.52	0.21
37	y	4616	55	6	45.99	45.21	5.24	1.1	2.25	0.19
38	func	4603	70	6	31.98	52.42	4.52	7.15	2.85	1.09
39	i	4575	79	6	79.93	13.01	2.86	1.53	1.49	1.18
40	for	4521	64	6	23.65	5.82	2.96	11.04	23.38	33.16
41	args	4454	78	6	32.24	51.8	5.68	2.9	7.09	0.29
42	file	4381	87	6	40.93	29.1	9.24	9.13	7.42	4.18
43	new	4331	81	6	66.66	20.18	5.7	2.89	3.86	0.72
44	forward	4071	36	6	3.05	2.14	4.03	15.82	73.96	1.01
45	start	3960	77	6	43.26	33.61	10.81	2.88	9.19	0.25
46	all	3927	79	6	66.62	6.32	8.58	6.67	11.31	0.51
47	index	3844	77	6	46.57	30.18	11.81	3.51	6.35	1.59
48	params	3799	62	6	41.64	29.27	7.29	2.45	18.79	0.55
49	field	3777	39	6	38.87	28.99	11.68	4	9.9	6.57
50	template	3715	46	6	59.84	19.78	11.6	2.91	3.82	2.05

Table 4.11: Most common words used in identifier names

4.9 Most common words

Among all the 745,651 identifier names investigated in this thesis, there are 1,421,977 words are used. As mentioned in Section 3.2, all the words used in the 100 repositories are converted to lower case. Table 4.11 shows the top 50 most common words. These words are used a total of 333816 times. This accounts for 23.48% of all the words used. As

Data types	Words	Number of uses	Examples
Numeric	integer	102	weight_integer
	int	757	allow_int_inputs
	float	238	read_float64
	double	124	is_double
String	string	1880	export_string
	str	1679	xml_str
Sequence	list	6571	vocab_list
	lst	107	lst_len
	array	936	ctype_ndarray
Mapping	dictionary	127	data_dictionary
	dict	5048	word_dict
Boolean	boolean	67	GetBooleanField
	bool	186	is_bool

Table 4.12: The uses of words related to built-in Python data types

mentioned in Section 4.8, some most common names are only used in specific repositories and domains. In contrast, all most all the 50 most common words are used in the all the 6 categories. Only the word `attention` is only existed in 3 categories which are AI, automation Tool and computer-vision.

Five of the most common words are single letters, which are `x, d, n, y and i`. These letters exist as separate words, such as the word `i` in the identifier name `plot_i_dynamic`. Words that contain the letter are not being discussed here, such as the letter `i` in the word `integer`. Two of top 50 most common words are numbers which are `1` and `2`. Most of the top 50 most common words are used in the names of variables, parameters and attributes. Of the identifier names that contain the word `i`, they have the highest percentage of single letter names. They are account for 70.2%. Followed by word `x`, which is 57.5%. Numbers are not allowed to use individually as identifier names.

Most of the top 50 common words shown in Table 4.11 are dictionary words. Excluding the seven single letters and numbers mentioned above, there are 8 abbreviations in the top 50 most common names. These words are `id`, `url`, `ids`, `num`, `dict`, `doc`, `func` and `args`. These are common abbreviations in programming. This means that programmers consider the concepts they refer to to be meaningful. Programmers are probably familiar with the common abbreviations. We believe that using these names might not cause comprehensibility problems.

We investigated some words related to the built-in Python data types. Table 4.12 shows some of the words and abbreviations that we think might be used to refer to data types. The table does not include acronyms. This is because most of the meanings they refer to may be unrelated to the data type. For example, the lowercase letter `l` is almost never used to refer to a list, and its most common application is to refer to the load in the file permissions read, load, and write. Another case is the use of `i` as an index, which is not intended to emphasise that the type is an integer. As mentioned in Section 3.2, this is just a common usage. We found that even in dynamic languages like Python, programmers use words related to types in their names. Among them, the most used are `list` and `dict` which are also occurred in the top 50 most used words. In contrast, it is rarely specified in the name that it is a numeric data type in Python repositories.

There should be more integers compared to dictionaries and lists in source code but there are relatively less words related to integers as shown in Table 4.12. We found that the choice to use abbreviations is more frequent when the type name is longer. For example, `dictionary`, `boolean` and `integer` are less frequent than `dict`, `bool` and `int`. The abbreviation `lst` is less likely to be used than `list`. This might be because the difference in length between the two words is not large enough. The abbreviation does not shorten the identifier name much in this case. However, the word `string` and its abbreviation `str` have similar number of uses.

We found that most of the common words are higher in variables, parameters, and attributes. This is because most identifiers belong to these three. However, there are also words that are used more in functions/methods, such as `get`, `to`, `is` and `set`. The words `get`, `set` and `is` are all verbs and are commonly used as the first word in function/method names. As shown in Table 4.13, among all functions/methods, 14.33% of their names use the word `get`, 2.85% use `set`, and 1.84% use `is`. The word `to` is a linking word which is mostly used in functions/methods related to change something and to-do something. For exmaple, `convert_to` in `freqtrade` and `apply_to_keypoint` in `albumentations`. Of the top 50 most common words in functions/methods, 24 are verbs, which is about half. We also found that in function/method names, words that are related to types are usually related to the types of the return values.

As shown in Table 4.14, we also investigated the top 50 most common words in class names. Compared to functions/methods, the repetition of words used in class names is relatively low. The most common word `model` is only used in 28 repositories. There are no particularly common words such as `get` in the class names. Even the word `model` is only found in 5.73% of the class names. Abbreviations and numbers are more common in class names. There are 12 of them in the top 50, and 7 of them are top 20 most common words. The words `exception` and `error` are only commonly used in class names. This is because that they are used in classes related to catch errors and print error messages. Other words related to built-in data types are not commonly used in class names. We noticed that the third commonly used word in the class is `tf`, which is not commonly used in other programming languages. This term is an abbreviation for `TensorFlow`, a library for machine learning and artificial intelligence, which is widely used in Python repositories in related fields. If a class name contains this abbreviation, it means that the

	Words	Number of Repetitions	Number of Repositories	Number of Categories	Usage %
1	get	16143	93	6	14.33
2	to	3671	82	6	3.26
3	forward	3655	31	6	3.25
4	set	3205	70	6	2.85
5	create	2557	71	6	2.27
6	call	2430	37	6	2.16
7	from	2330	74	6	2.07
8	update	2214	67	6	1.97
9	is	2068	80	6	1.84
10	data	1956	58	6	1.74
11	validate	1873	52	6	1.66
12	add	1857	71	6	1.65
13	check	1833	71	6	1.63
14	extract	1659	44	6	1.47
15	list	1575	62	6	1.4
16	for	1556	58	6	1.38
17	on	1545	45	6	1.37
18	load	1401	68	6	1.24
19	name	1334	59	6	1.18
20	type	1326	56	6	1.18
21	function	1267	37	6	1.13
22	delete	1205	36	6	1.07
23	gen	1138	29	6	1.01
24	handle	1110	48	6	0.99
25	real	1092	17	6	0.97
26	model	1082	34	6	0.96
27	output	1051	51	6	0.93
28	make	1004	59	6	0.89
29	id	995	46	6	0.88
30	run	991	63	6	0.88
31	init	989	53	6	0.88
32	build	951	52	6	0.84
33	key	942	56	6	0.84
34	value	906	52	6	0.8
35	embeddings	904	9	3	0.8
36	process	892	61	6	0.79
37	and	886	63	6	0.79
38	config	883	50	6	0.78
39	input	880	38	6	0.78
40	2	835	51	6	0.74
41	convert	834	51	6	0.74
42	params	807	41	6	0.72
43	execute	792	24	6	0.7
44	dict	784	60	6	0.7
45	custom	748	28	6	0.66
46	parse	740	59	6	0.66
47	batch	725	27	6	0.64
48	file	725	71	6	0.64
49	state	716	44	6	0.64
50	apply	709	45	6	0.63

Table 4.13: Most common words - Function/Method

class might have an inheritance relationship with `TensorFlow`.

	Words	Number of Repetitions	Number of Repositories	Number of Categories	Usage %
1	model	1548	28	6	5.73
2	for	1499	13	6	5.55
3	tf	1157	7	6	4.29
4	error	1054	50	6	3.9
5	ie	1010	2	4	3.74
6	2	816	31	6	3.02
7	layer	692	16	6	2.56
8	base	672	55	6	2.49
9	flax	667	1	1	2.47
10	classification	595	6	3	2.2
11	bert	584	6	1	2.16
12	config	577	32	6	2.14
13	pre	552	3	6	2.04
14	output	550	19	6	2.04
15	lm	506	4	3	1.87
16	exception	504	24	6	1.87
17	data	474	44	6	1.76
18	not	464	26	6	1.72
19	encoder	452	18	5	1.67
20	attention	446	9	3	1.65
21	trained	430	2	2	1.59
22	t	427	14	6	1.58
23	meta	406	24	6	1.5
24	invalid	397	17	6	1.47
25	conv	325	11	6	1.2
26	type	325	40	6	1.2
27	net	316	12	6	1.17
28	head	310	9	5	1.15
29	sim	306	4	6	1.13
30	v	303	21	6	1.12
31	d	296	27	6	1.1
32	mixin	290	22	5	1.07
33	migration	289	7	4	1.07
34	found	283	21	6	1.05
35	sequence	272	11	6	1.01
36	vi	268	4	5	0.99
37	tokenizer	263	5	3	0.97
38	serializer	260	8	4	0.96
39	text	254	28	6	0.94
40	token	253	15	6	0.94
41	view	253	17	6	0.94
42	field	248	16	6	0.92
43	question	244	1	3	0.9
44	decoder	237	10	4	0.88
45	group	232	29	6	0.86
46	self	230	3	6	0.85
47	answering	228	1	1	0.84
48	backend	225	15	6	0.83
49	image	223	23	6	0.83
50	set	219	21	6	0.81

Table 4.14: Most common words - Class

5 Discussion

In the previous chapter, we have discussed the results of the 9 naming practices separately. In this chapter, we are going to discuss the connection between these naming practices and answer the 3 research questions (Section 3.1).

Overall, We can find some poor naming practices in the source code that are difficult to understand, such as using single letter names for classes (Section 4.1). However, there are also some naming practices that do not follow some general naming conventions but are relatively easy to understand. For example, using some common abbreviations (Section 4.8).

5.1 How are the different naming conventions used in practice?

We investigated the prevalence of 9 naming practices in the Python repositories. Among them, 4.26% are single letter names, 1.67% are names ending with numbers, 0.49% have numbers in the middle of the names, 69.03% are composed of dictionary words, 63.56% of the function/method names are verb phrases and 77.6% of the class names are noun phrases. In summary, for most parts, the choices of identifier names are consistent with the guidelines.

With these results, we believe that programmers deliberately follow the general suggestions from the naming conventions in most cases.

In Section 2.2, we mentioned the use of numbers in identifier names can be categorised into 6 types. They are arbitrary numbers, distinguishers, version numbers, synonyms and special uses (specification and domain/technology) [45]. Among them, there are no arbitrary numbers in either of the 2 naming practices related to numbers. Arbitrary numbers can not describe the concept of identifiers since they are meaningless. Programmers might waste time trying to understand the meaning. Our results show that programmers probably agree that the guidelines for not using meaningless numbers are correct. We think they recognise that it is good to follow such guidelines, and this is why they choose to do so.

Except for the use of arbitrary numbers, the 2 naming practices related to numbers are being used differently in the source code. For number at end, the numbers are more likely to be used as distinguishers in variables, parameters and attributes. However, for number in middle, there is a preference for specific uses and synonyms. These results show that there is a difference in their uses. Number at end is used more to distinguish similar concepts, and number in middle is used to shorten the length of names.

Among the length-related naming practices investigated in this thesis, it is shown that class names are the longest and they consist of the largest number of words. Variables, parameters and attributes are relatively shorter and they also have a higher percentage of single letter names (Section 4.1). This might relate to the complexity of the concepts of identifiers. The concepts of classes and functions/methods are usually more complex compared to other identifiers. We believe that in order to accurately describe these more complex concepts, naming conventions should be developed in a way that allows for the use of more wordy names. The results also show that programmers generally accept the use of longer names to name classes.

There are naming conventions that suggest the use of dictionary words for identifiers [28]. 69.03% of the names are composed entirely of dictionary words. This might be because the use of some common abbreviations does not cause confusion. Many words that are not dictionary words also appear in the results of the top 50 for most common words and most common names. This means that programmers are familiar with these words. The names are in their knowledge base that are easier to understand and remember.

In the study by Gresta et al. [23], they mentioned that the indiscriminate use of the same names for different concepts in one repository might cause misunderstanding. In our results, only 9 of the 50 most common names were composed of two words, and the rest were names of single words. This shows that the same names rarely occur when describing relatively complex concepts. Relatively simple concepts are more likely to have the same name, probably because these concepts themselves occur frequently in software. For example, `name, x, value, config, and data` are the 5 most common names. These concepts may not require additional modifiers other than the nouns themselves.

We investigated the names with the words that may be associated with identifier types in Python code. The results show that in dynamic programming language, programmers would also choose to store the type-related information in the identifier names. This might mean that the types of these identifiers will not be changed in subsequent code. We also found that there are fewer uses of acronyms compared to full names and abbreviations of types. We investigated letters related to types and found that they are usually not related to types, but rather acronyms of other words. The study by Gresta et al. [23] also mentions that the use of single letter names related to types is less common in Java and C++ than other single letter names, although it does not investigate the use of type-related words in names. The study also mentioned that the use of type-related names may increase the reading load. We believe that the use of type-related words in identifier names in Python may be different. The types of identifiers in Java or C++ can be found in the declaration lines, but there is no declaration step in Python. Therefore, we think doing this can emphasis the types of identifiers.

In this thesis, we investigated grammatical structures related to noun structures and verb structures. The results show that programmers prefer to use verb structures for naming functions and methods. Other identifiers (variable, parameter, attribute, class) are usually noun structures. We found cases where verbs are omitted from function/method names, and also cases where nouns are omitted from classes names and only modifiers are retained.

In the case of omitting verbs, if the verb `get` is omitted, we consider its impact on comprehensibility to be small. This practice is common in the code we studied, so we think programmers might be used to doing it. However, we think other cases are relatively confusing. In particular, in the case of omitting nouns, it is difficult to infer the object of the modification from the adjective itself. Moreover, there are no nouns in class names can be used in same way as `get` in functions/methods. This means that there are no words that programmers can immediately react to when they read a class name without a noun. There are variables, parameters, attributes, and classes in the code that are named as verb phrases. We believe that this would cause comprehensibility problems because programmers might think the concepts of the identifiers containing actions.

In addition to suggesting the use of verb structures for naming functions/methods, Abebe et al. [1] suggest that function/method names should start with a verb. We believe that while this might be able to emphasis the actions contained in function/method names. It is not really necessary, as the concepts of functions/methods are relatively complex and require modifiers. This means that there are cases where adverbs are used to modify the concepts before the verbs. The programmers still need to read the whole names to fully understand the concept described by the identifier names.

5.2 How much influence does context have on the choice of identifier names?

We investigated the differences between 9 naming practices in different contexts. The different contexts are identifiers used in statements (If, While, For) and scope length. Our results show that most of the naming practices are different in different contexts. This means that programmers might choose different identifier names depending on the context of the identifiers.

By investigating naming practices related to length, we found that identifiers used in statements (If, While, For) are shorter and have a higher percentage of using single letters as identifier names. Our results also show that longer identifier names are more likely to be used in larger functions/methods. This suggests that in some cases, the concepts of identifiers are simpler or already partially contained in the knowledge base. For example, the use of i as an index in range type for loop statements. In these cases, programmers might commonly use less descriptive identifier names which are relatively shorter.

Apart from length, only the usage of number in middle is related to the scope length of functions/methods. Our results show that in larger functions/methods, programmers tend to use more identifier names that contain numbers in the middle. Usually in larger scopes, the concepts of identifiers are more complex, thus requiring the use of longer names. As we mentioned earlier in Section 5.1, number in the middle is used to shorten the length of the name. We think this might mean that programmers intentionally shorten names to improve reading speed and comprehensibility. Our results show that how parameters are named does not change with respect for the scope length of functions/methods.

We found that the naming of functions and methods is similar in single letter, number at end, number in middle, grammatical structure. However, the results show that methods have a higher percentage of using dictionary words than functions and also method names are relatively shorter than function names. We believe this may be related to their scope. Since the scope of the method is a class, the concepts of methods might be simpler than concepts of functions in a larger scope. This might because for method names, there is no need to mention the information already contained in the class names. Part of the concepts of the methods are already contained in their classes. This means that the concepts of methods can be accurately described by using shorter names, and there is no need to shorten the names by using abbreviations.

Our results also show that the naming of variables and attributes are similar in all the context and repository properties. However, parameters are different for both of them in some cases. In Section 4.1, we mentioned that almost all the single letter `x`'s are used as parameters. In our results, using single letter `x` is much less common in other kinds of identifiers. There is also relatively less use of names with numbers in the parameter names.

Among all the 3 types of statements (If, While, For), the identifier names in If statements are the longest. For loop and While loop are both loop statements, but the results show that their names are also different. The range type for loop statements have the shortest names, with 84.3% of them using only one word. They also have the highest percentage of single letters. On the other hand, the 3 types of identifiers are similar in the use of dictionary words and grammatical structures.

5.3 Is there a difference in the choice of names in different Python repositories?

We investigated 5 repository properties, which are LoC, number of commits, number of contributors, number of releases and the first release date. The statistical results show that only some of the number of commits, number of contributors, and number of releases correlate with the naming practices.

Both number at end and number at middle are more likely to be used in larger repositories. This might because the concepts of identifiers are more complex in larger repositories, which also leads to the possibility that they might need to be differentiated by using numbers or reduce the length of the identifier names. In Java and C++ repositories, number at end has the same trend, but it shows there is no relationship between number in the middle and LoC [24].

In smaller repositories, identifier names are shorter and use fewer words. This also shows the evidence of the concepts of identifiers are usually more complex in larger repositories, so they need more words to be accurately described. Our results show that the first release is not related to the length of identifier names, but is related to number of words used in the names. This means that identifier names in older repositories use more words in identifier names, but the length of each word is shorter than in newer repositories. On average, abbreviations are shorter than dictionary words because they are used to shorten the length of the words. Therefore, our results indicate that older repositories might contain more identifier names with abbreviations. This could also be used to demonstrate that programmers have become more aware of using dictionary words for identifier names in the recent 10 years.

Of the 4 types of data related to dictionary words `All, At least one, All but one, and None` that we investigated for dictionary words, only `All but one` is related to repository properties. Programmers prefer to use names that contain only one non-dictionary word in small repositories. `All but one` also contains cases where there is only one word and it is an abbreviation. We noted that single letter names are not related to these two repository properties. This could mean that they are related to single non-acronym abbreviations.

For repository categories, our results show that they are related to most naming practices. We noted that in scientific-computing repositories, the identifier names are relatively short and use fewer words. This is also consistent with results in other naming practices. The percentage of using verb phrases for naming functions/methods in scientific-computing is relatively lower which is only 49.37%. The results show that the practice of omitting verbs is more common in scientific-computing compared to other categories. This might because there are more functions/methods that are used to return a certain thing which usually starts with `get`. Single letter, number at end and number in middle are also more common in this category. The most common single letters used in this category are `x`, `a` and `i`. Letter `i` is usually used as an index. Letters `x` and `a` are used for unknown values. This can only be found in scientific-computing repositories.

Noun phrases are the same across categories. That is, no matter what the categories are, programmers think that they should use noun phrases to name class names.

Another category that uses more identifier names containing numbers is computer-vision. This category contains many identifier names related to 2D and 3D axes. This means that these repositories would have many identifier names like `x1, x2, y1, y2` to distinguish between different points on the axis. The results for single letter also show that `x` and `i` are the most common single letter names. Letter `y` is also used more than any other categories.

We think this implies that the programmers' knowledge base on name choices varies across domains. This also needs to be considered when giving naming suggestions.

6 Conclusion

In this chapter, we are going to summarise the achievements of this empirical study which investigated the use of Python identifier names and also some related future work.

6.1 Contributions

In this thesis, we have described how to use the Python AST package to build a tool for extracting and examining the information of identifiers from source code. With this tool, we investigated the use of 745,651 identifier names in 100 open source python repositories in different context and the repository properties. To answer our research questions, we have selected and discussed 9 naming practices in 5 areas. They are number-related, length-related, type-related, syntactic structures and common words.

This thesis is inspired by Gresta et al.'s study [23, 24] of the association of naming practices with context in Java and C++. In Chapter 2, we have introduced and discussed the related work in software comprehension and also the definition of proper meaningful identifier names.

We have introduced the 3 research questions in Section 3.1 which are also being listed below:

RQ1: How are the different naming conventions used in practice?

RQ2: How much influence does context have on the choice of identifier names?

RQ3: Is there a difference in the choice of names in different Python repositories?

We have presented and discussed the results in Chapter 4 and Chapter 5. For research question 1, our results show that, programmers are deliberately following the naming conventions for most of the time. For research question 2 and 3, our results show that the use of identifier names does vary across cases. The choice of identifier name can be influenced not only by context but also by repository properties. We believe that this may be related to the different information contained in the programmer's knowledge base in different contexts, which may already contain information that partially describes the concepts of the identifiers.

As we mentioned in Section 5.3, method names are usually shorter than function names. This might be the partial concepts related to methods overlap with the classes they belong to. There would be no need to repeat it again in method names. This means that in this case it might not be necessary to describe this part of the information in the identifier name. Therefore, making more detailed naming conventions based on different cases can help programmers to choose names that more accurately describe the concept of identifiers and thus improve comprehensibility.

Our results also show that although Python is a dynamic programming language, it has similarities with other static programming languages (such as Java and C++) in the choice of identifier names. Some general naming conventions are common between these languages, but might still require some more subdivided differences depending on the different cases. For example, using single letter `i` as a loop counter and using distinguishers as a pair of parameters should be allowed.

6.2 Future Work

This thesis focuses on the Python identifier names. We believe that this research can be extended to other dynamic programming languages (such as PHP and JavaScript) as well as static programming languages (such as Java, C++, and C#). By comparing the differences between these languages, it might be helpful to suggest more detailed and appropriate naming conventions.

In this thesis, we can only analyse the source code to find out some patterns of identifier names, but it is not possible to know exactly why the programmers made these choices. Therefore, we believe that a user study involving experienced programmers is necessary. Through this kind of study, it might be possible to know the reasons why programmers choose not to follow certain guidelines in some cases. This might help to the suggest naming conventions that are better suited to certain cases.

We believe that the results of this thesis can also contribute to Python-related technical studies in the field of software comprehension. For example, the development of a tool that can give proper naming suggestions depending on different contexts and domains. This could help programmers to choose more appropriate names and thus improve comprehensibility.

At the end, we hope that the conclusions of this thesis will contribute to more Python-related research in the software comprehension field.

A
Appendix

Repository	Single letter		Number at end		Number in middle		Dictionary word		Verb structure (function/method)		Noun structure (class)		Length	Number of words
	n	percentage %	n	percentage %	n	percentage %	n	percentage %	n	percentage %	n	percentage %	average	average
activitywatch	0	\	0	\	0	\	59	93.65	2	40	0	\	11	2.11
AidLearning-FrameWork	20	2.75	30	4.12	0	\	508	69.78	48	41.74	11	84.62	9.25	1.88
AITemplate	1057	4.69	345	1.53	360	1.6	11976	53.13	1363	40.14	297	71.91	10.6	2.11
albumentations	291	7.01	131	3.16	3	0.07	2250	54.22	520	73.45	105	75.54	8.57	1.83
angr	1990	4.78	1064	2.55	191	0.46	25123	60.28	4179	63.12	1261	74.62	8.87	1.79
arrow	6	1.43	10	2.39	0	\	296	70.64	35	39.33	0	\	7.79	1.5
AugLy	117	4.2	16	0.57	1	0.04	1990	71.4	280	71.98	48	46.15	10.57	1.96
augmented-traffic-control	21	2.99	5	0.71	0	\	453	64.53	110	79.14	0	\	8.42	1.6
autoscraper	2	0.91	2	0.91	0	\	154	70	25	83.33	0	\	8.66	1.71
avatarify-python	38	8.82	6	1.39	0	\	269	62.41	45	75	8	72.73	7.39	1.55
BayesianOptimization	40	9.78	0	\	0	\	289	70.66	40	45.45	14	93.33	8.11	1.48
BERTopic	24	2.44	0	\	0	\	775	78.92	86	84.31	23	76.67	9.54	1.64
borg	86	1.72	39	0.78	19	0.38	3796	75.98	558	66.91	134	64.42	8.7	1.68
Cactus	28	2.19	11	0.86	2	0.16	1001	78.39	187	58.81	55	90.16	8.93	1.6
catalyst	160	1.39	28	0.24	7	0.06	8797	76.18	1131	58.57	362	70.84	10.03	1.79
cli	15	0.74	25	1.24	1	0.05	1567	77.77	258	64.82	98	89.09	10.12	1.76
corona	15	23.08	0	\	3	4.62	31	47.69	4	66.67	0	\	4.66	1.26
dangerzone	11	1.3	1	0.12	0	\	674	79.67	145	78.8	37	67.27	11.35	1.9
dash	44	2.44	8	0.44	0	\	1355	75.24	205	68.11	48	88.89	10.52	1.86
deeplake	275	2.3	79	0.66	13	0.11	8822	73.84	1401	67.45	246	83.11	10.18	1.89
django	413	1.35	143	0.47	98	0.32	23990	78.2	3680	59.33	1336	80.05	10.2	1.79
dotbot	1	0.33	0	\	0	\	272	90.67	44	73.33	13	86.67	8.01	1.33
eht-imaging	1096	6.91	1941	12.24	190	1.2	6170	38.89	367	32.33	26	81.25	6.64	1.85
erpnext	963	2.88	56	0.17	0	\	28576	85.43	5700	88.36	542	82.62	11.63	2.02
espnet	733	7.06	64	0.62	69	0.66	5294	50.99	518	52.32	255	86.73	8.02	1.78
evidently	77	1.53	42	0.83	4	0.08	3813	75.58	539	69.46	351	86.45	11.24	1.94
EyeWitness	42	9.84	1	0.23	0	\	270	63.23	32	36.36	0	\	7.96	1.58
face_recognition	0	\	0	\	0	\	111	77.62	14	56	0	\	12.43	2.14
fairlearn	93	9.13	10	0.98	0	\	627	61.53	75	60.48	14	73.68	10.72	1.95
fastapi	5	0.54	6	0.65	4	0.43	718	77.45	80	62.99	101	95.28	10.89	1.81
FATE	975	3.29	303	1.02	155	0.52	21496	72.57	3424	71.42	1095	88.88	10.58	1.96
feast	52	0.64	34	0.42	5	0.06	5786	70.95	847	57.82	263	82.97	12.19	2.11
FeelUOwn	111	3.06	36	0.99	2	0.06	2761	76	426	56.42	276	85.45	8.83	1.68
FlagAI	1009	6.51	332	2.14	110	0.71	10030	64.71	991	50.25	484	81.34	9.56	1.88
flair	156	2.3	32	0.47	11	0.16	5175	76.37	627	60.17	340	85.86	11.14	1.98
FlareSolverr	20	3.42	1	0.17	2	0.34	441	75.51	69	56.56	19	95	9.25	1.65
flask	51	4.19	0	\	0	\	895	73.6	169	56.71	40	90.91	9.06	1.65
flet	158	0.9	369	2.09	1	0.01	14903	84.55	867	34.89	258	86	13.38	2.21
FlexGen	2502	1.94	1354	1.05	1108	0.86	91796	71.02	8783	61.57	4031	64.13	12.44	2.2
flexx	355	9.6	133	3.6	5	0.14	2502	67.68	442	65.68	123	93.18	7.35	1.52
frappe	508	2.35	24	0.11	5	0.02	17713	82.01	4015	82.97	418	83.94	10.06	1.83
freemocap	83	2.91	15	0.53	81	2.84	1963	68.85	344	76.11	53	92.98	15.58	2.78
freqtrade	119	1.32	50	0.56	4	0.04	7076	78.65	1064	65.68	268	88.16	10.38	1.92
geemap	62	2.4	32	1.24	1	0.04	1802	69.74	198	53.23	18	85.71	9.35	1.78
genshin_auto_fish	115	5.12	91	4.05	5	0.22	1205	53.67	132	56.65	45	91.84	7.71	1.71
geopy	23	1.63	27	1.91	2	0.14	930	65.77	59	24.89	48	78.69	8.5	1.5
Gerapy	18	1.8	2	0.2	9	0.9	830	83.17	75	50	37	97.37	8.34	1.48
Gymnasium	326	4.94	160	2.43	10	0.15	4706	71.37	667	64.76	153	63.75	9.25	1.71
html2text	27	10.59	0	\	3	1.18	173	67.84	15	34.88	0	\	8.47	1.74
HTTPretty	12	2.09	5	0.87	0	\	422	73.65	49	51.58	16	88.89	9.34	1.72
imgaug	353	3.17	448	4.02	29	0.26	7570	67.93	1016	64.84	211	63.94	9.45	1.84
jupyter-book	1	0.38	0	\	0	\	166	62.41	23	58.97	0	\	9.42	1.74
keras	831	3.72	235	1.05	145	0.65	16126	72.26	2577	65.13	422	77.86	10.69	1.89
legit	8	3.38	0	\	0	\	185	78.06	28	56	3	75	7.95	1.41
lightfm	10	2.49	0	\	0	\	324	80.6	43	76.79	0	100	10.86	1.81
LinkFinder	2	2.74	0	\	0	\	58	79.45	3	37.5	0	\	8.16	1.55
lora-scripts	6	5.45	0	\	0	\	72	65.45	17	70.83	0	\	8.41	1.64
manim	241	3.71	229	3.53	38	0.59	4284	66.03	1061	74.88	170	70.54	9.67	1.93
memray	1	0.24	4	0.98	2	0.49	348	84.88	60	82.19	29	96.67	9.54	1.68
Meshroom	155	3.37	5	0.11	20	0.43	3784	82.23	567	66.32	159	84.57	9.17	1.71
ml-stable-diffusion	32	2.57	22	1.77	14	1.13	835	67.12	57	52.29	22	78.57	11.45	2.2
moco	10	17.54	0	\	0	\	23	40.35	1	16.67	2	66.67	8.19	1.82
moto	182	0.37	124	0.25	69	0.14	38868	78.71	6745	74.9	1522	75.27	12.61	2.06
neural-style	5	3.65	4	2.92	2	1.46	93	67.88	8	42.11	0	\	8.77	1.62
noisy	0	\	0	\	0	\	34	62.96	13	92.86	1	50	9.19	1.7
numpy	2432	13.35	688	3.78	237	1.3	10204	56.01	1521	49.71	284	82.56	7.06	1.61
OpenBBTerminal	383	1.61	172	0.72	29	0.12	17358	72.8	2742	82.17	173	89.64	9.13	1.69
osmnx	114	7.19	33	2.08	0	\	923	58.2	100	55.56	0	\	8.88	1.76
PandasGUI	88	4.87	24	1.33	0	\	1396	77.21	181	56.39	60	89.55	8.47	1.69
pelican	46	3.31	0	\	8	0.58	1146	82.39	161	74.54	40	95.24	8.98	1.61
pyforest	4	2.67	1	0.67	0	\	103	68.67	18	72	0	\	11.29	1.95
pyinfra	14	0.36	5	0.13	19	0.49	3126	79.91	467	76.18	185	91.58	9.13	1.62
pyTelegramBotAPI	8	0.88	0	\	0	\	814	89.75	151	71.23	49	80.33	8.12	1.62
pytesseract	0	\	0	\	0	\	116	71.6	15	57.69	2	33.33	8.45	1.52
pyxel	222	20	48	4.32	1	0.09	636	57.3	87	63.97	0	\	7.64	1.72
ralph	45	0.39	29	0.25	30	0.26	9383	80.44	1221	71.87	1456	91.63	11.14	1.87
requests	30	3.12	14	1.46	0	\	730	75.88	106	59.22	40	90.91	8.42	1.54
scdl	0	\	0	\	0	\	10	66.67	1	50	0	\	7.33	1.47
SciencePlots	0	\	0	\	0	\	3	50	0	\	0	\	12	2
scipy	6906	21.71	1553	4.88	162	0.51	13011	40.9	1320	32.28	315	87.02	5.79	1.49
sh	29	3.78	3	0.39	0	\	470	61.2	102	68.46	19	82.61	8.7	1.72
SiamMask	296	8.07	240	6.55	7	0.19	1934	52.76	163	48.66	70	95.89	6.73	1.59
sktime	3097	14.08	574	2.61	21	0.1	11811	53.7	2008	73.53	406	84.23	8.68	1.72
SlowFast	422	7.58	121	2.17	22	0.4	3284	58.98	360	58.44	88	79.28	8.96	1.88
SonoffLAN	13	1.13	35	3.05	1	0.09	588	51.31	100	44.84	50	79.37	9.71	1.89
SpaceshipGenerator	15	4.52	0	\	0	\	252	75.9	22	84.62	0	\	9.48	1.87
spaCy	123	2.31	406	7.63	42	0.79	3498	65.73	524	69.22	74	90.24	8.27	1.67
streamalert	0	\	4	0.12	9	0.27	2774	82.51	543	67.96	136	86.08	11.15	1.8
s-tui	18	2.33	1	0.13	0	\	582	75.29	110	70.51	24	92.31	10.72	2
Sublist3r	34	7.2	3	0.64	0	\	348	73.73	68	81.93	13	81.25	7.99	1.54
sweetviz	33	3.86	17	1.99	1	0.12	623	72.87	80	75.47	9	81.82	11.19	2
tinygrad	302	12.21	32	1.29	14	0.57	1456	58.85	219	42.03	107	84.25	6.52	1.49
toapi	1	1.27	0	\	0	\	61	77.22	5	29.41	0	\	6.3	1.22
tpot	50	5.62	20	2.25	0	\	572	64.34	93	69.4	18	94.74	10.18	1.84
visdom	67	6.89	5	0.51	9	0.92	625	64.23	114	65.9	20	71.43	6.93	1.49
web3.py	20	0.51	147	3.77	3	0.08	2665	68.26	559	62.81	165	84.62	11.77	1.99
whoogle-search	11	2.73	0	\	0	\	255	63.28	33	55	0	\	9.01	1.7
WireViz	10	2.63	5	1.32	3	0.79	248	65.26	27	46.55	0	\	8.24	1.62
word_cloud	34	5.7	12	2.01	1	0.17	405	67.95	58	72.5	0	\	8.65	1.66
youtube-dl	692	2.78	90	0.36	221	0.89	17675	71.13	1148	42.49	1098	84.2	8.43	1.67

Bibliography

[1] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. Lexicon bad smells in software. In *2009 16th Working Conference on Reverse Engineering*, pages 95–99. IEEE, 2009.

[2] Reem Alsuhaibani, Christian Newman, Michael Decker, Michael Collard, and Jonathan Maletic. On the naming of methods: A survey of professional developers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 587–599. IEEE, 2021.

[3] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. A new family of software anti-patterns: Linguistic anti-patterns. In *2013 17th European conference on software maintenance and reengineering*, pages 187–196. IEEE, 2013.

[4] Eran Avidan and Dror G Feitelson. Effects of variable names on comprehension: An empirical study. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 55–65. IEEE, 2017.

[5] Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G Feitelson. Meaningful identifier names: The case of single-letter variables. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 45–54. IEEE, 2017.

[6] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I Maletic, Christopher Morrell, and Bonita Sharif. The impact of identifier style on effort and comprehension. *Empirical software engineering*, 18:219–276, 2013.

[7] Dave Binkley, Matthew Hearn, and Dawn Lawrie. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 203–206, 2011.

[8] Dave Binkley, Dawn Lawrie, Steve Maex, and Christopher Morrell. Identifier length and limited programmer memory. *Science of Computer Programming*, 74(7):430–445, 2009.

[9] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983.

[10] Simon Butler, Michel Wermelinger, and Yijun Yu. A survey of the forms of Java reference names. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 196–206. IEEE, 2015.

[11] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 156–165. IEEE, 2010.

[12] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Mining java class naming conventions. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 93–102. IEEE, 2011.

[13] C Caprile and Paolo Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*, pages 112–122. IEEE, 1999.

[14] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 560–564. IEEE, 2021.

[15] Brad Dayley. *Python phrasebook: essential code and commands.* Sams Pub., 2007.

[16] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14:261–282, 2006.

[17] Dmitri Pavlutin. Coding like shakespeare: Practical function naming conventions. `https://dmitripavlutin.com/coding-like-shakespeare-practical-function-naming-conventions`. Online; accessed 6 December 2023.

[18] Len Erlikh. Leveraging legacy system dollars for e-business. *IT professional*, 2(3):17–23, 2000.

[19] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 26th Conference on Program Comprehension*, pages 286–296, 2018.

[20] Dror G Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. How developers choose names. *IEEE Transactions on Software Engineering*, 48(1):37–52, 2020.

[21] Martin Fowler and Kent Beck. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.

[22] James Gosling. *The Java language specification.* Addison-Wesley Professional, 2000.

[23] Remo Gresta, Vinicius Durelli, and Elder Cirilo. Naming practices in java projects: An empirical study. In *Proceedings of the XX Brazilian Symposium on Software Quality*, pages 1–10, 2021.

[24] Remo Gresta, Vinicius Durelli, and Elder Cirilo. Naming practices in object-oriented programming: An empirical study. *Journal of Software Engineering Research and Development*, pages 5–1, 2023.

[25] Guido van Rossum, Barry Warsaw, Alyssa Coghlan. Pep 8 – style guide for python code. `https://peps.python.org/pep-0008/`. Online; accessed 10 December 2023.

[26] Samir Gupta, Sana Malik, Lori Pollock, and K Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 3–12. IEEE, 2013.

[27] Emily Hill, Lori Pollock, and K Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *2009 IEEE 31st International Conference on Software Engineering*, pages 232–242. IEEE, 2009.

[28] Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)*, pages 217–227. IEEE, 2017.

[29] Honnibal, Matthew and Montani, Ines and Van Landeghem, Sofie and Boyd, Adriane and others. spacy: Industrial-strength natural language processing in python. `https://spacy.io/`. Online; accessed 22 October 2023.

[30] Einar W Høst and Bjarte M Østvold. The java programmer's phrase book. In *International Conference on Software Language Engineering*, pages 322–341. Springer, 2008.

[31] Dawn Lawrie, Henry Feild, and David Binkley. Syntactic identifier conciseness and consistency. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 139–148. IEEE, 2006.

[32] Dawn Lawrie, Henry Feild, and David Binkley. Extracting meaning from abbreviated identifiers. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 213–222. IEEE, 2007.

[33] Dawn Lawrie, Henry Feild, and David Binkley. Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12:359–388, 2007.

[34] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *14th IEEE international conference on program comprehension (ICPC'06)*, pages 3–12. IEEE, 2006.

[35] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3:303–318, 2007.

[36] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4):325–339, 1987.

[37] Ben Liblit, Andrew Begel, and Eve Sweetser. Cognitive perspectives on the role of naming in computer programs. In *PPIG*, page 11, 2006.

[38] Jonathan I Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 103–112. IEEE, 2001.

[39] Robert C Martin. *Clean code: a handbook of agile software craftsmanship.* Pearson Education, 2009.

[40] George A Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[41] Christian D Newman, Reem S AlSuhaibani, Michael J Decker, Anthony Peruma, Dishant Kaushik, Mohamed Wiem Mkaouer, and Emily Hill. On the generation, structure, and semantics of grammar patterns in source code identifiers. *Journal of Systems and Software*, 170:110740, 2020.

[42] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. Evaluating code readability and legibility: An examination of human-centric studies. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 348–359. IEEE, 2020.

[43] Michael P O'brien. Software comprehension–a review & research direction. *Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report*, 2003.

[44] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341, 1987.

[45] Anthony Peruma and Christian D Newman. Understanding digits in identifier names: An exploratory study. In *Proceedings of the 1st International Workshop on Natural Language-based Software Engineering*, pages 9–16, 2022.

[46] Dusty Phillips. *Python 3 object oriented programming*. Packt Publishing Ltd, 2010.

[47] Python Software Foundation. Abstract Syntax Trees. `https://docs.python.org/3/library/ast.html`. Online; accessed 17 September 2023.

[48] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265. IEEE, 2012.

[49] Giuseppe Scanniello and Michele Risi. Dealing with faults in source code: Abbreviated vs. full-word identifier names. In *2013 IEEE International Conference on Software Maintenance*, pages 190–199. IEEE, 2013.

[50] Andrea Schankin, Annika Berger, Daniel V Holt, Johannes C Hofmeister, Till Riedel, and Michael Beigl. Descriptive compound identifier names improve source code comprehension. In *Proceedings of the 26th Conference on Program Comprehension*, pages 31–40, 2018.

[51] Yusuke Shinyama, Yoshitaka Arahori, and Katsuhiko Gondow. Improving semantic consistency of variable names with use-flow graph analysis. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, pages 223–232. IEEE, 2021.

[52] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8:219–238, 1979.

[53] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*, pages 174–188. 2010.

[54] Jeremy Singer and Chris Kirkham. Exploiting the correspondence between micro patterns and class names. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 67–76. IEEE, 2008.

[55] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14:187–208, 2006.

[56] Armstrong A Takang, Penny A Grubb, Robert D Macredie, et al. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

[57] Barbee E Teasley. The effects of naming style and expertise on program comprehension. *International Journal of Human-Computer Studies*, 40(5):757–770, 1994.

[58] TIOBE organization. Tiobe index for november 2023. `https://www.tiobe.com/tiobe-index`. Online; accessed 29 November 2023.

[59] Nicole Vavrová and Vadim Zaytsev. Does python smell like java? tool support for design defect discovery in python. *arXiv preprint arXiv:1703.10882*, 2017.