Allegheny College
Department of Computer Science

Senior Thesis Proposal

NamePy: An Identifier-Focused Linter for Beginning Python Developers

by

Thomas Antle

ALLEGHENY COLLEGE

COMPUTER SCIENCE

Project Supervisor: **Douglas Luman**
Co-Supervisor: **Oliver Bonham-Carter**

29 July 2022

Abstract

Programmatic identifiers are used by software developers to assign names to entities in their programs. They provide the foundation of program comprehension and readability, so it is highly important that they are written clearly. There are not many naming conventions that are required by programming languages, so developers or communities create clear names and standards on their own. Programmers develop this skill as they gain more experience, but the difficult is high for students. This research proposes a Python tool that examines and evaluates identifiers in students source code to help them learn this skill. It will be implemented at the continuous integration level for ideal results. This research will also include an evaluation of the tool via a human study. The study will use participants at Allegheny College with prior Computer Science experience to evaluate the effectiveness of the tools user interface. Conclusions will be drawn based on participant answers compared to the tools intended solutions.

Introduction

Statement of the problem

Creating identifiers when writing programs is an important and necessary part of every developers job. As opposed to many other words used in source code, identifiers are created by the developer, not preset by libraries or programming languages. This fact in itself creates a task that must be addressed. Identifier names must be created and thought-over with care. Developers should think thoroughly about their naming conventions in accordance with several factors; thus where a problem arises. Identifiers that are not thoughtfully named will likely cause difficulty for third party readers of the code. In addition, clear identifiers that are accurately descriptive are instrumental in debugging code as well as increasing parsing efficiency. Factors that need to be addressed include but are not limited to identifier length, style, comprehension and readability. This proposal will focus on the idea of developing better identifiers in accordance with naming conventions that are not required in the Computer Science field but are rather centered on good hygiene in writing code.

Overall aim

This research project aims to create a tool in Python that evaluates identifier names in Python source code. The criteria used for this evaluation will be determined based on the research that is outlined in the next section. Overall, the tool will:

- Be leveraged by Python users
- Output messages for specific identifier errors
- Rate programs according to their identifiers

With the help of this tool, developers will be given feedback that helps to improve readability and code comprehension by naming identifiers in a better way.

Background motivating research

This project idea initialized with my background in the Computer Science field starting as a beginner. There is much to learn at the beginning stages, so naturally there are some topics that are not taught to the fullest degree. For me, one such topic was naming conventions. Snake case, camel case, numbers, letters, identifier content and identifier types are all things that remained uncertain for me until I gained more experience as a student. As a student developer I am pursuing this project to aid other developers in the struggles that I experienced. On a personal level I am pursuing this project to learn more about the complexities of programmatic naming conventions, but also naming and language in a broad sense. Aside from my personal background, initial research into the area proved to offer more motivation as well. Information about identifier length, style, grammar and associated comments as well as the topics of reading comprehension and linting all aided with my creation and understanding of this projectt goals.

Related Work

There are several aspects to consider when determining an adequate identifier name. It is necessary for it to be long enough to be descriptive, but short enough for comprehension. It is necessary for it to use part-of-speech grammar, but it is not certain what grammar is most effective. It is necessary for it to use a style type that is required by the program, but a style type is not always required. It is not always necessary to include descriptive comments, but there could be a correlation with comprehension. It is necessary for it to be easily comprehended, but many of the previously mentioned topics could affect its ability to do so. There is no correct or perfect way to name identifiers due to personal preference, but this research provides a way for developers to enforce a standard that will yield overall better results.

Background of Identifiers

An identifier in a programming language is the term used to describe lexical tokens that name various things in a program (variables, functions, etc.). To create names for identifiers, it is important to view them in terms of language. Derek Jones makes the connection that There are a number of situations in everyday life that require the use of arithmetic and binary concepts (e.g., switch light on/off, change mind and flag an error), and measurements confirm that words and phrases commonly used in human conversation are carried over into identifier names [6]. With this in mind, identifiers can be thought of in a largely similar way to natural language. This paves the way to looking at topics such as word length, grammar and comprehension when thinking about identifier creation. In addition to this, Annette Vee makes the distinction that Computers are also socially contextualized; they are objects that are both controlled by language and can be used to manipulate linguistic symbols [18]. This defines identifiers as the lexical tokens that they

are. This fact is what makes identifiers unique to programming as opposed to natural language. So, while it is important to view them as a type of language themselves, it is also important to remember their purpose in code: to symbolize.

Identifier Length

Binkley et al. present a study that looks at short-term memory in relation to identifier names. The study was conducted in Java rather than Python, but nonetheless it remains relevant as a study conducted in a similar programming language. Several aspects of identifier names were seen to have affected the participants comprehension. One finding showed that more syllables equated to longer processing times and worse recall accuracy [4]. This shows that in general, longer identifiers (more syllables often equates to longer words) will take longer and be more difficult to comprehend. This contradicts the idea of making identifiers as descriptive as possible because they may end up too long at times. Rather, it suggests that they need to be as short as possible without compromising how descriptive they are. Interestingly, it was also found that longer name lengths had greater influence on experienced programmers [4]. This suggests that the necessary length of identifiers for better comprehension may change as a programmer becomes more experienced.

The use of common identifiers was also seen to improve recall accuracy with the participants of the study [4]. This shows that a larger amount of uniformity in identifier names (using more common words that are used often in the community as a whole) improves comprehension. More will be discussed about uniformity in the following sections.

Schankin et al. present another study that draws conclusions about longer, more informative identifier names. They state that Code comprehension was assessed by measuring the time needed to find a semantic defect in the code, assuming that such a defect can only be detected if the code has been understood. It was found that (particularly with more experienced developers) there was better comprehensibility with more descriptive names. In contrast to Binkleys study, it does not highlight the importance of keeping names concise. Similar to the previous study, developer experience has an affect on how short an identifier can be made while maintaining descriptiveness [12].

Another study performed by New et al. deals with a slightly different but fundamentally related topic: lexical decision. Using the English Lexicon Project, they reexamined the effect of word length (number of letters in a word) on lexical decision [7]. Binkleys study that was previously mentioned focused on short term memory (shortening identifiers) while Schanklins study focused on long term memory (lengthening identifiers). Compared to both of these, this study found that the effect of word length was not clear [7]. Because both studies suggested different things, an unclear result justifies that contrast. These findings suggest that ideal identifier length is fluid according to various aspects. Given this, the tool created will not be able to suggest very specific standards regarding length, but rather it will lend guidance in a broader sense.

Identifier Style

Davis et al. (also authored by Binkley as we have seen before) present an interesting study about identifier style in relation to effort and comprehension. The two main styles are examined in the study (camel case and snake case) in various ways to determine comprehensibility. The main takeaway of the article is that camel case yields better comprehension at the cost of more effort [3]. This project is focused in Python, so styles are determined by the official style guide. For example, class names should be typed with the **CapWords** convention and functions names should be typed with the **snake_case** convention [11]. Though the articles findings do not specifically matter for the purpose of this project, they prove a point of uniformity. The fact that there is a difference in comprehension between styles at all proves that using a uniform style in a program has value. Essentially, if Python developers had the freedom to use various styles in various places as they saw fit, the code would be less comprehensive in general and less readable to developers that follow the standard. Davis et al. give one piece of evidence here by showing the difference in comprehension between just two styles. It is clear that reading code with various styles (that are each easier or harder to comprehend) would result in code that is overall much harder to comprehend. The main takeaway is that uniformity is an important characteristic of identifier style, and thus it may be important relating to other identifier topics too.

Identifier Grammar

Newman et al. present an investigation into identifiers in relation to the parts-of-speech used in them. The study found various results that could relate to this project. One such finding was that Function identifiers are more likely to contain a verb and be represented by a verb phrase [8]. This makes sense due to the fact that functions have actions and verbs can adequately describe actions. This foundation of identifiers describing things based on their role (for instance a function completing an action) is key to creating understandable identifiers. This can be seen if, for instance, a function identifier were named with a noun. It would not adequately describe the function and therefore not be very comprehensive. This finding also shows that grammar patterns are naturally seen even with no direct convention required when creating identifiers. The fact that certain grammar conventions more adequately describe certain identifier types as well as the fact that these conventions are often seen naturally shows that a uniformity of grammar conventions is beneficial. Similar to uniformity with identifier style, maintaining consistent grammar conventions would aid with comprehension.

Associated Comments

Takang et al. present a study that presents a new, rather overlooked idea when looking at identifiers: comments describing those identifiers. The results yield a surprisingly low correlation between program comprehension and identifiers being paired with comments. One potential reason for this was stated that if the comments did not provide any new information that is not already conveyed in the identifier names, then the effect would be insignificant considering that exactly one of them would do just as well [16]. This reinforces the importance of making identifiers descriptive. Comments are not a technical part of source code, rather they are an aid for readers. As with any aid, they should not be relied on consistently. Rather, the identifiers should be relied on for code comprehension and therefore they should be the descriptive ones that make comments superfluous.

Song et al. present the fact that some researchers have tried to define identifiers with a long descriptive name in order to implement self-commented code. However, it makes code comprehension more difficult [15]. It is true that there are practical limits to identifier lengths and therefore limits to

how descriptive they can be. Because of this, a combination of descriptive identifiers and descriptive comments is the best answer.

Reading Comprehension

Schoeman presents an eye-tracking study that involves reading skills in relation to programming. Her results showed that there is a direct correlation between reading skill and program comprehension. Given that younger students are generally those that lack in reading skills, the study would be mostly relevant to student developers. One main takeaway is that program flow is not sequential, requiring a substantial degree of selective attention, organizing and reflecting, executed simultaneously in the limited working memory [13]. This idea is helpful when thinking about identifiers as well. It is essential to know that the reader is going through a complex flow in their head when trying to understand the program as a whole (identifiers included). This makes it even more important for identifiers to fit in the program and flow smoothly by having the best names possible.

Annette Vee has an exert in her book that reads Like reading, [programming] is comprised of a large number of abilities that interrelate with the organization of the learners knowledge base, memory and processing capacities, repertoire of comprehension strategies, and general problem-solving abilities. As reading is often equated with skill in decoding, learning to program in schools is often equated with learning the vocabulary and syntax of a programming language. But skilled programming, like reading, is complex and context-dependent [18]. This exert adds onto the previous point that there is a complexity associated with programming similar to that of linguistic aspects of reading. Since the processes are similar, there is justification in saying that ability to read and comprehend written text has bearing on ability to read and comprehend programs. These connect via literacy according to Vee. This will be discussed in the `Coding as Literacy` section.

Given these points about reading comprehension, it should be possible to develop a readability score for identifiers in a similar way to scoring written text. By incorporating the other mentioned topics, a score could be further elongated to provide better and more descriptive score results.

Linting

Linting originates from a Unix utility which examines C source programs, detecting a number of bugs and obscurities. It was used as a tool to enforce the rules of C more than the compiler alone [5]. From the original C linter to todays Python linters, the fundamental purpose has not changed. In Python, Linting highlights syntactical and stylistic problems in your Python source code, which often helps you identify and correct subtle programming errors or unconventional coding practices that can lead to errors [1]. Essentially, linters are intended to catch errors in order to improve source code. The aim of this project is to do the same thing with identifiers. Poorly named identifiers will be caught and the user will have a better understanding of how to make them better. Given this, the methods outlined in the next section focus on linting at various levels in regard to identifiers.

Another term that is often used as a replacement for `linter` is `static analysis tool`. A list that is comprised of the widely used static analysis tools for every programming language can be found at this GitHub repository. Included in this list are three linters that hold high reputations and most closely resemble NamePy. These tools are `pylint`, `flake8` and `pycodestyle`. Being Python linters, they are intended to catch syntactical and stylistic errors in source code. It is important to note, though, that these tools (including NamePy) are largely suggestive in nature. The use of linters is not technically required in any instance unless otherwise specified by a manager of a project. PEP8 (the official Python style guide) notes that A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important [11]. This quote suggests that the importance of linting lies in the aspect of consistency. In this way, while linters are not required, they are encouraged due to the consistency that they bring. Ultimately, the creators of linting tools are able to choose the style guide that they follow. This may include rules from PEP8 as well as rules that are not official according to the Python language. Despite the rules (as mentioned before) the importance is the consistency that the tool brings. `pylint`, `flake8` and `pycodestyle` are comprised of style guides that many Python developers agree with and depend on. For this reason, NamePy will be compared to them in following section.

Method of Approach

Feature Comparison

The following table displays the differences in features between this proposals tool NamePy and three other popular linters. It defines the feature set that will be used with NamePy and compares it to them in order to show both similarities and singularity.

Feature	NamePy	pylint	flake8	pycodestyle
Ability to Run in GitHub Actions	✓	✓	✓	✓
Enforce PEP8 Naming Conventions	✓	✓	✓†	✓
Check Identifier Length	✓			
Check Identifier Grammar	✓			
Check for Identifiers w/ Associated Comments	✓			
Error Evaluation Score	✓	✓		
Comprehension Score	✓			

- † Only with optional PEP8 plugin

The feature comparison table highlights the main features of NamePy next to `pylint`, `flake8` and `pycodestyle`. The ability to run a linter in GitHub Actions is a feature that will be discussed in this section. It is an important feature for any linter to have, so it is present in all four. The official style guide of Python (PEP8) contains a section for naming conventions that is vital for NamePy. Before any other features are addressed, abiding by PEP8 naming standards is the key focus of the tool. The other three linters in the table all abide by all or some PEP8 standards as well. Various topics mentioned in the `Related Works` section will be implemented as features

including identifier length and grammar checks as well as associated comment checks. Because these three features analyze identifiers in a way that research suggests is effective, they are unique to NamePy. Finally, an evaluation score based on the amount of errors in a program is a feature that `pylint` possesses, but not `flake8` or `pycodestyle`. Though, NamePy will have another evaluation score based on identifier reading comprehension that none of the other tools have.

Feature	NamePy
Check Identifier Length	✓
Check Identifier Grammar	✓
Check for Identifiers w/ Associated Comments	✓
Comprehension Score	✓

The table above highlights once more the unique features of NamePy. While other linters focus on PEP8 standards as one of their main features, they are seen as a default on this tool (only the naming conventions of PEP8). Rather, the unique features are the main focus that separate this tool from others. Checking for identifier length will consist of setting a limit as to how short is too short and how long is too long based on research studies from `Related Works`. Checking for identifier grammar will consist of ensuring that certain identifiers do not have improper parts of speech. For instance, function names should use verbs while variable names should not. Checking for associated comments with identifiers will consist of doing just that. Lastly, a comprehension score will split identifiers into readable words for existing comprehension tests to run them and produce scores.

Coding as Literacy

Annette Vee describes code in her book *Coding Literacy* in a way that lays the framework for my chosen methods in this project. She links code and literacy into one by stating that programming is a literacy itself. While the term literacy has an extensive history with many different definitions over time, it has often been and often is associated with reading and writing. Linking this to coding, Vee states that Since the advent of text-based programming languages, code is a form of writing as well as an enactment of procedures [18]. By accepting the fact that coding (reading and writing the code) is

a literacy, cognitive theories of writing can be applied to it. In this way, arguments and approaches regarding literacy and writing can be leveraged to determine the methods for this project. This topic will be further discussed in the next sections.

Levels of Implementation

IDE Level

IDEs (Integrated Development Environments) are software applications that are leveraged by most developers to manipulate source code. Given the popularity and versatility of Visual Studio Code, I am choosing to use it to discuss linting at the IDE level. VSCode extensions is a large feature that gives the software application a lot of its versatility. Users are able to quickly and simply search for and install extensions that can be found on the extension marketplace.

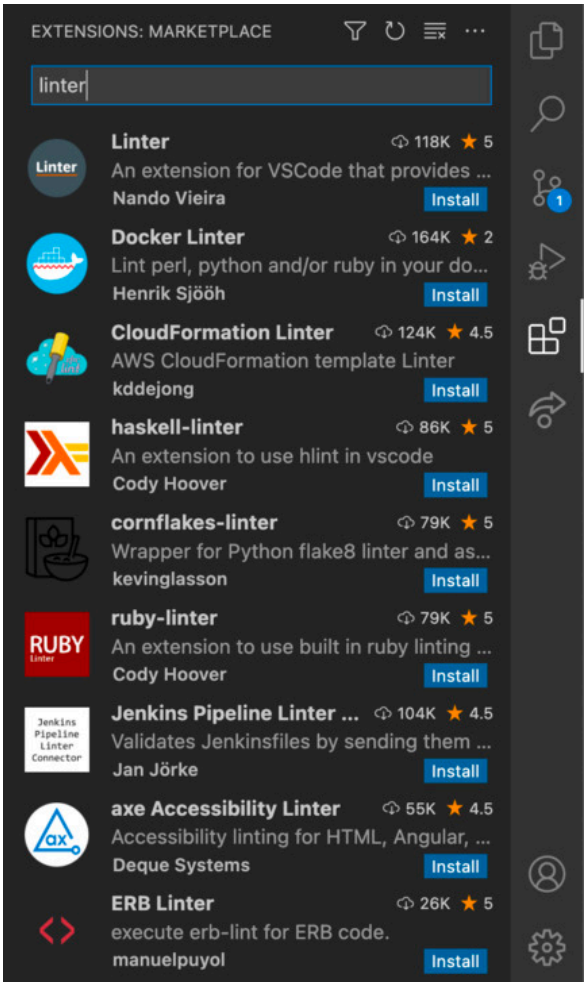

Figure 1: Extension Marketplace on Visual Studio Code

What is beneficial about linting at this level is the very short length of time between making an error, identifying and fixing it. A participant in a study performed by Tomasdottir et al. stated that If you can get some bugs away from your code so early as when you write it, its great [17]. The rigidity of this concept can be seen by the sheer amount of VSCode extensions that exist for the purpose of catching errors as they are written. By shortening the time between making a mistake and finding/fixing it, the hardship of fixing them when they are found during compile or runtime is decreased [17].

Considering lower-level, junior developers, it is important to look at how

catching errors at the IDE level affects learning. In general, corrective feedback is proven to be very useful in terms of acquiring further cognitive skills. Obermuller et al. also states how positive feedback is considered to have better effects on motivational aspects than negative feedback [9]. Linters are inherently negative/corrective in nature, so they will always have some sort of negative impact on student motivation. Though, IDE errors appear in a very slow, one-by-one fashion compared to linters at the CI level. This suggests minimal negative impact on student motivation while maintaining corrective feedback to acquire further cognitive skills.

Despite the above information, the IDE level is not the ideal level to apply the tool with students due to cognitive complexity. When comparing the time of linting between the IDE level and a separate level (such as GitHub Actions), it can be seen that there is a higher cognitive complexity at the time of the IDE. This is due to the fact that the given task at the IDE level is to write code. By adding a linter at the same time, the difficulty of that task is increased due to the fact that errors are presented at the time of writing. Robinson presents in his article that The effects of complexity differentials should be revealed by the fact that the cognitively simpler, less resource-demanding task will involve a lower error rate, and/or be completed faster, and be less susceptible to interference from competing tasks than the more complex task [10]. This is important due to the fact that students will have a higher error rate given higher cognitive complexity when coding. Since the goal of the tool is to help students improve their source code, implementing it in a way that is too complex may hurt them instead.

CI Level

While the IDE level of linting possesses many tools that developers use to address feedback, this projects tool is intended for students, so evaluation is more important than formative feedback. IDE level linting is not normally capable of providing evaluation metrics, so a different route would be required. Implementation of the tool in a similar manner as `Pylint` for instance (the ability to run the linter in a VSCode window upon saving) would provide an evaluation, but this integration does not fit the scope of the project. Rather, the use of GitHub Actions is a better choice to provide evaluative feedback to student developers.

Referring back to Vee, in addition to the fact that coding is a literacy, she also enforces the fact that literacy is developed (or learned) and not

inherent. Suggesting both that literacy is a learned skill and that this is especially prevalent in students, Vee states that Sophisticated literacy skills such as analysis and argument have always been necessary at the highest educational echelons, but we now expect all students to achieve this level of skill under the rubric of literacy [18]. This is an important connection to make considering the evaluative nature of my tool. Student developers require evaluation on their work because they are constantly learning.

Keeping in mind that coding is a literacy, Sommers provides insight based on his research on student writers. He found that students often did not revise their work because they lacked a set of strategies to help them identify thesomething larger that they sensed was wrong and work from there [14]. In the same way, IDE level visualization of errors do not always help students learn how to revise their code. Rather, evaluation metrics at the GitHub Actions level would help students learn their mistakes and correct them in a fixing something larger manner. This is what makes the CI level optimal for this application.

GitHub Actions

Identical to popular linters such as `Pylint` the tool can be inserted into GitHub Action workflows for automation. This does not require any additional implementation to the tool itself. To do so, a `.yml` file can be created that installs and runs the tool on the given files/directories inside of a virtual environment. GitHubs documentation provides information for any developer to be able to implement the tool in this manner. Including the tool in a class of students continuous integration build would be beneficial because it would set a standard of linting that would be followed by the whole class. Compared to running it locally, putting the tool in a build workflow would take away the manual labor and therefore make it easier and more useful. For student developers this is also a good option because instructors can create workflows incorporating it. In this way students do not have to do any extra work and they are able to automatically see errors as they appear in GitHub.

Local Use

Like other open source linters, this tool will have the capacity to be installed locally on machines and to be run with software projects that students work on. It will be available on PyPI for users to install with the `pip` command

and it will have the capacity to run in a similar manner to other linters such as `Pylint`. This method of using the tool is the most basic for student developers. Given that it is the same process as other popular linters, it is expected that any user will know how to use it given the documentation. Though it is the most basic, it does not offer any sort of automation. In order to check for linting errors, the user would need to manually run the tool every time they wished to check their work. Initial setup of the tool is the most simplistic compared to the previous two levels, but continued use of it is the most time-demanding. It will remain as a standard way to use the tool for any user that wishes to do so, but leveraging GitHub Actions is recommended.

LibCST

The tool for this project will leverage LibCST as its main library to parse source code and extract information regarding identifiers. While one would normally use Pythons built-in AST module to leverage syntax trees and parse them, LibCST has more to offer. Essentially, it creates a compromise between an Abstract Syntax Tree (AST) and a traditional Concrete Syntax Tree (CST) [2]. Unlike traditional syntax trees, this library is lossless in the sense that it has the ability to preserve all parts of the source code including things like comments, whitespace and parenthesis.

The main framework for my tool stems from a previous project that I contributed to named CAStanet. There is a file that allows it to parse through either individual Python files or entire Python directories. A command line interface exists to perform various actions on the files. These features will also be used for this tool. At the root, the tool uses `matchers` to locate specific source code items such as assignment statements, function definitions and comments. Using this ability, various functions will be created to locate identifiers and analyze them. The tool will look for specified criteria in all of the identifiers and develop a score based on how well the criteria were met. It will also report back to the user specific errors with identifiers in a similar way to other popular linters.

```
def var_length(path:str):
    cast_dict = file_or_directory(path)
    for cast in cast_dict.values():
        var_list = match.findall(cast, match.Assign())
```

The LibCST library provides the groundwork to easily extract necessary information from programs for running functions that meet the tools linting and scoring goals.

Evaluation Strategy

The evaluation of this project will be tested with with other Python programs. Because NamePy is intended for students, programs of varying levels (from beginner Python users to more experienced ones) will be leveraged. With these programs, NamePy will be run on them in order to find identifier errors in the code.

Choosing the programs that I will test the tool on will be a process that is entirely unbiased. It is not certain what results will be yielded when the time comes to choose the programs, so there are two potential methods that exist in order to guarantee adequate and unbiased code to use. The first method will be to access open source Python programs on GitHub. To find projects that are of the students level and of various student experience, public Allegheny College organizations would be examined. Given that there are enough public repositories containing Python source code from different level of Computer Science courses, programs will be randomly chosen from the lot of them. It is important when selecting programs for a test like this that they are entirely random. To do this, once the organizations are found, every student project will be assigned a number. A random number generator will then choose which 1-3 projects to test with NamePy. It is also important that the particular experience level of each program is not known upon the time of testing. To keep this hidden, each program will be randomly numbered rather than named so that no bias can come from running them. Once all testing is complete, the results will be recorded and the experience levels will be revealed.

Upon the potential finding there are not adequate, public repositories within Allegheny College Computer Science course organizations, the aid of one or multiple professors will be used instead. In this case, the professors will create Python programs that are progressive in the same way real student growth would be. The programs would then be tested without any knowledge

of each level of program. At the end of testing when results are seen, the levels would be revealed and conclusions would be drawn from them.

The results to this evaluation will look like scores as well as sets of numbers. Given that the tool outputs a an evaluation score based on the amount of errors versus the amount identifiers, the score can be very easily used to compare programs. In addition to score, the number of errors can be closely monitored as well. The output of errors and scores will ultimately determine progression across the programs and will determine validity of NamePy.

The goal of these methods is to determine the validity of NamePy. The results should show that low-level programs output a high number of identifier errors than higher level programs. In the same way, a graph with the number of errors should show a progression in experience level. This will show that the tool is helpful to both students and educators. As a student, it will be known that the tool will find errors and as an educator it will be known that the tool will help with teaching. In the case that the results do not suggest validity in this way, the tool will be reflected on to see where any issues lie. If the issues are feasible to fix in a short amount of time, they will be. In any case, the thesis would include the reflection on those issues and discuss what could have been changed.

Research Schedule

Date	Task
07/22	Proposal Defense
07/23	Begin Two Draft Chapters
07/23	Begin Development
08/09	Two Draft Chapters Due
10/17	Complete Development
10/17-10/31	Test Tool
12/02	Complete Thesis
12/05-12/07	Thesis Defense
12/08-12/20	Incorporate Suggested Thesis Changes
12/20	Final Deadline

Conclusion

This project proposes a tool that looks into Python identifiers in both a linguistic and a technical manner. Evaluating the identifiers of student developers is a process that resembles that of reading and writing. Being that identifiers are an artifact of literacy, concepts of reading and writing apply in how I view them for this project. This has impacted my conceptual decisions as well as the way I am approaching my methods and evaluation of the tool. To apply all of this, technical specifications have also been thought through in terms of viewing identifiers and making the tool work. This project aims for the tool to be implemented like other popular linters in the development process in order to produce more productive identifiers. The idea comes from the lack of standardized naming in the Python language as well as the difficult task given to junior developers.

As a finished product that is executed correctly, the tool should give students evaluation metrics that suggest how to improve the identifiers in their programs. It should also instill beneficial habits that lead to quality name creation in the future. My intent is for this projects tool to be used by students who are learning to improve their source code. Instructors of beginner Computer Science students or students themselves may wish to leverage it as an additional linter to promote good naming practices. I believe that the results of this project will provide aid and insight to help junior Python users improve their habits and skills.

References

[1] Linting python in visual studio code. Retrieved from `https://code.visualstudio.com/docs/python/linting`

[2] Why LibCST? Retrieved from `https://libcst.readthedocs.io/en/latest/why_libcst.html`

[3] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering* 18, (April 2013). DOI:https://doi.org/10.1007/s10664-012-9201-4

[4] Dave Binkley, Dawn Lawrie, Steve Maex, and Christopher Morrell. 2009. Identifier length and limited programmer memory. *Science of Computer Programming* 74, 7 (2009), 430445. DOI:https://doi.org/`https://doi.org/10.1016/j.scico.2009.02.006`

[5] S. C. Johnson. 1978. Lint, a c program checker. (1978). Retrieved from `https://wolfram.schneider.org/bsd/7thEdManVol2/lint/lint.pdf`

[6] Derek M. Jones. 2008. Operand names influence operator precedence decisions. *C Vu* 20, 1 (February 2008), 511.

[7] Boris New, Ludovic Ferrand, Christophe Pallier, and Marc Brysbaert. 2006. Re-examining word length effects in visual word recognition: New evidence from the english lexicon project. *Psychonomic bulletin & review* 13, (March 2006), 4552. DOI:https://doi.org/10.3758/BF03193811

[8] Christian D. Newman, Reem S. AlSuhaibani, Michael J. Decker, Anthony Peruma, Dishant Kaushik, Mohamed Wiem Mkaouer, and Emily Hill. 2020. On the generation, structure, and semantics of grammar patterns in source code identifiers. *Journal of Systems and Software* 170, (2020), 110740. DOI:https://doi.org/https://doi.org/10.1016/j.jss.2020.110740

[9] Florian Obermüller, Lena Bloch, Luisa Greifenstein, Ute Heuer, and Gordon Fraser. 2021. Code perfumes: Reporting good code to encourage learners. (October 2021). DOI:https://doi.org/10.1145/3481312.3481346

[10] P Robinson. 2001. Task complexity, task difficulty, and task production: exploring interactions in a componential framework. *Applied Linguistics* 22, 1 (March 2001), 2757. DOI:https://doi.org/10.1093/applin/22.1.27

[11] Guido van Rossum, Barry Warsaw, and Nick Coghlan. 2001. PEP 8 - the style guide for python code. (2001). Retrieved from https://peps.python.org/pep-0008/

[12] Andrea Schankin, Annika Berger, Daniel V. Holt, Johannes C. Hofmeister, Till Riedel, and Michael Beigl. 2018. Descriptive compound identifier names improve source code comprehension. (2018), 3140. DOI:https://doi.org/10.1145/3196321.3196332

[13] Marthie Schoeman. 2019. Reading skills can predict the programming performance of novices: An eye-tracking study. *Perspectives in Education* 37, 2 (2019), 3552. Retrieved from https://www.proquest.com/scholarly-journals/reading-skills-can-predict-programming/docview/2446291563/se-2?accountid=8268

[14] Nancy Sommers. 2003. Revision strategies of student writers and experienced adult writers. In *Cross-talk in comp theory: A reader*, Victor Villanueva Jr. (ed.). National Council of Teachers of English (NCTE), Urbana, Ill., 4354.

[15] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. 2019. A survey of automatic generation of source code comments: Algorithms and techniques. *arXiv.org* (2019). Retrieved from `https://www.proquest.com/working-papers/survey-automatic-generation-source-code-comments/docview/2264511570/se-2`

[16] Armstrong Takang, Penny Grubb, and Robert Macredie. 1996. The effects of comments and identifier names on program comprehensibility: An experimental investigation. *J. Prog. Lang.* 4, (September 1996), 143167.

[17] Mauricio; van Deursen Tomasdottir Kristin Fjola; Finavaro Aniche. 2017. Why and how JavaScript developers use linters. (2017), 578589. DOI:https://doi.org/10.1109/ASE.2017.8115668

[18] Annette Vee. 2017. *Coding literacy: How computer programming is changing writing.* The MIT Press. Retrieved from `http://www.jstor.org/stable/j.ctt1s476xn`