Bachelor's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Amalie Nystuen
Kristina Ødegård

Qualities of Professional Python Code

Bachelor's thesis in Dataingeniør
Supervisor: Ali Alsam
June 2023

NTNU
Norwegian University of
Science and Technology

Amalie Nystuen
Kristina Ødegård

Qualities of Professional Python Code

Bachelor's thesis in Dataingeniør
Supervisor: Ali Alsam
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Programming proficiency extends beyond mere knowledge of programming language syntax. Novice programmers often lack problem-solving skills demonstrated by experts, who employ a variety of schemata and patterns to categorize problems effectively. This distinction emphasizes the importance of mastering problem-solving skills to attain proficiency in programming. Educational models illustrating professional source code characteristics are vital for guiding beginner programmers to become experts (Windslow, 1996).

In this research project, we analyzed and visualized the qualities of professional Python code. The dataset comprises 30 widely used open-source libraries with a total of 11,712 source code files. To facilitate the analysis, each file was transformed into nodes using Python's Abstract Syntax Tree (AST). The nodes were then quantified, resulting in a 229-dimensional vector representation for each file. To explore the dataset, we utilized organizational structures from two programming textbooks. Guided by these structures, we generated informative visualizations with educational value.

Principal Component Analysis (PCA) and k-means clustering analysis revealed that the professional source code in the dataset can be seen as one cohesive unit. The findings from the data analysis indicate *functions* as the primary building block of professional code, with widespread use of object-oriented programming. Furthermore, the code is structured into concise blocks like *functions*, *loops*, and conditions. By incorporating these insights into programming courses, we aim to contribute to the improvement of instruction quality and facilitate students' progress towards proficiency.

Samandrag

Evna til å programmere strekk seg utover kunnskap om syntaksen i programmeringsspråk. Nybyrjarar i programmering manglar ofte dei problemløysingsevnene som ekspertar demonstrerer ved å nytte seg av ei rekke skjema og mønster for å effektivt kategorisere problem. Denne skilnaden framhevar viktigheita av å ha gode problemløysingsevner for å oppnå ferdigheiter på ekspertnivå. Modellar for undervisning som illustrerer kvalitetar ved profesjonell kjeldekode, er avgjerande for å rettleie nybyrjarprogrammerarar til å bli ekspertar (Windslow, 1996).

I dette forskingsprosjektet har vi analysert og visualisert kvalitetar ved profesjonell Python-kode. Datasettet består av 30 mykje brukte open-source bibliotek, bestående av totalt 11 712 filer. For å lette analysen, vart kvar fil omgjort til nodar ved hjelp av Python-modulen Abstract Syntax Tree (AST). Nodane vart deretter kvantifisert, noko som resulterte i ein 229-dimensjonal vektorrepresentasjon for kvar fil. For å utforske datasettet, brukte vi strukturen frå to lærebøker i programmering. Ved hjelp av desse strukturane, genererte vi informative visualiseringar med pedagogisk verdi.

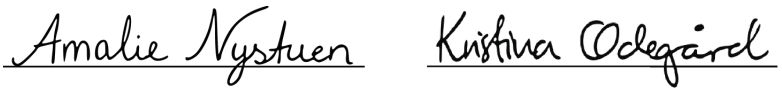
PCA og k-means-klyngeanalyse viser at den profesjonelle kjeldekoden i datasettet kan bli sett på som ei samanhengande eining. Resultata frå dataanalysen indikerer at funksjonar er ein viktig byggestein i profesjonell kode, og at bruken av objektorientert programmering er utstrekt. Vidare er kode strukturert i korte blokker, som funksjonar, løkker og vilkår. Gjennom å innarbeide desse funna i programmeringsundervisning, har vi som intensjon å forbetre kvaliteten på undervisninga og lette studentane sin framgang mot ekspertise.

Preface

This bachelor thesis is written by two computer science students at The Norwegian University of Science and Technology (NTNU) during the spring of 2023. The thesis was written on behalf of our supervisor and client, Ali Alsam; Associate Professor at Department of Computer Science.

The motivation behind selecting this task stemmed from the opportunity to make a contribution to programming education, by exploring the qualities of professional Python code. The research project provided us with the avenue to explore diverse aspects and possibilities within the field, while also allowing us to apply the knowledge acquired from courses such as Big Data (INFT2003), Applied machine learning with project (IDATT2502), and Statistics (ISTT1003). The reader is assumed to have a foundational understanding of computer science. However, prior knowledge on the topic is not required.

We would like to express our sincere gratitude to our supervisor, Ali Alsam, for suggesting such an engaging and thought-provoking project, as well as for providing continuous support and guidance throughout the entire process. Furthermore, we extend our appreciation to the developers of the 30 open-source libraries analyzed in this thesis.



Amalie Nystuen Kristina Ødegård

Trondheim, June 8th 2023

Task description

The original task description was as follows:

Exploring the qualities of professional Python source code

The process of teaching programming involves guiding students on their journey to learn how to write conditions, iterations, functions, and classes. However, the common approach of teaching "how to program" often lacks instructions on best practices, and teachers may only be aware of a limited set of best practices.

In this project, our objective is to analyze a substantial collection of professional Python code extracted from well-documented open-source libraries. By doing so, we aim to explore the best practices employed by experienced programmers and visualize these practices in a manner that can serve as an educational aid in classrooms.

To extract the necessary metrics, we will represent each code file as a set of feature counts that can be analyzed and visualized. These features will be based on Python's AST, where each feature corresponds to a node in the tree. Examples of metrics include the percentage usage of functions, classes, iterations, and various data types.

Contents

List of Figures

List of Tables

1 Introduction

The ability to solve problems is an important indicator of a programmer's competence level. Although knowledge of the specific syntax of a programming language, such as C, C++, Java or Python, is essential, it is not sufficient to distinguish between beginners and experts (Windslow, 1996). Most programming languages have a limited set of keywords, e.g. the C++ language has 97 words (cpprefeence.com, n.d.), and Java has 67 reserved words (Oracle Corporation, 2021). In contrast, the Norwegian language has over 300 000 words, and English has over 500 000 registered words (Språkrådet, n.d.). Thus, if programming skills were judged by the ability to memorize the keywords, then it would be sufficient to know 35 words to claim that one is a proficient Python programmer (Python Software Foundation, 2021b).

The distinction between expert programmers and beginners is not limited to their knowledge of syntax. Rather, when experts attempt to categorize problems, they utilize a wide range of schemata and patterns, which beginners lack (Windslow, 1996). In addition, studies show that it is challenging for students to know "where and how to combine statements to generate the desired result (Windslow, 1996)."

Beginner programmers have superficial knowledge, and need to master problem-solving before they become proficient programmers. Further, research shows that the time required to progress from a beginner to an expert programmer is approximately ten years (Windslow, 1996)

To aid beginners on their path to becoming expert programmers, models are necessary for students to develop an in-depth understanding of the qualities of experts' source code (Windslow, 1996). Linn and Clancy (1992) recommend verbal descriptions, illustrations, and connections to real-world problems to aid the development of such educational models.

Our research project aims to identify and visualize the characteristics of professional Python source code to serve as an educational model that aids the progress of programming students. To achieve this goal, we utilize a dataset of professional Python code. Specifically, we analyze a dataset consisting of open-source code from 30 well-known libraries, including Matplotlib and Pandas. The files from the libraries adds up to a total of 11,712 files. For more information about the libraries, refer to Section 3.3.

To process the data, we employ a program developed by the research project initiator, Ali Alsam. This program utilizes the Python module AST to count feature variables, and generates a matrix with 11,712 rows and 229 columns. Each row corresponds to a file from the libraries, and each variable is represented as an element in a 229-dimensional vector. These variables encompass 176 counts of AST nodes, 35 Python keyword frequencies, 15 length counts, and counts of the function calls *input*, *print*, and *open*.

Our approach involves identifying common patterns and characteristics in the source code, including code organization and the use of data structures. To start with, we analyzed the dataset to explore whether it can be treated as a one cohesive unit. To this end, we employed PCA, and k-means clustering. The

findings from this analysis reveals that the dataset is in 181-dimensional space, and cannot be clustered in more than two groups, efficiently.

To guide our data analysis and visualizations, we examine the structure of two popular programming textbooks: "Python for Everyone" (Horstmann and Necaise, 2019) and "Starting out with Python" (Gaddis, 2019). This analysis of programming textbooks, coupled with the numerical data extracted from the open-source libraries, aims to provide data that can guide students on their path to becoming expert programmers. For each main chapter in the studied textbooks, we generate a set of visualizations and data analysis based on the exploration of the dataset from the libraries. For example, we explore the use of logical operators, data structures, loops, conditional statements, functions, error handling, and object-oriented programming.

Overall, our results indicate that professional programmers primarily use functions as the main building blocks of their code. Furthermore, the dataset demonstrates extensive use of object-oriented programming. We have also identified some less common practices in our analysis, such as classes defined within functions and within *if* statements.

By providing visualizations that depict the qualities of professional Python code, our aim is to enhance the quality of programming education.

1.1 Research questions

The purpose of the research project is to explore professional Python code to uncover the qualities and best practices for programming in Python. Visualization of these findings can contribute to improvement of programming education. The research questions are as follows:

- ***Can the dataset, which consists of 30 different libraries, and 11,712 files, be considered as one coherent set?***
- ***What are the qualities of professional Python code?***
- ***Can the qualities of professional Python code be used to guide the education of beginner programmers?***

1.2 Document structure

The report is divided into nine chapters with the following headlines and content:

- **Introduction** - This chapter introduces the bachelor thesis, outlining the background, research questions, report structure, and acronyms associated with the project.
- **Theory** - This chapter presents relevant theoretical information for the project, including a literature review.
- **Method** - This chapter describes the methodology and scientific process used throughout the project..
- **Results** - This chapter presents the results obtained from the analysis, including visualizations.
- **Discussion and Conclusion** - This chapter reflects upon the obtained results, and concludes by providing answers to the research questions.
- **Further work** - This chapter presents potential further work related to the project.
- **Societal Impact and Sustainability** - This chapter presents societal impact and sustainability related to the work.
- **References** - This chapter contains all references used throughout the report.
- **Attachments** - This chapter include all relevant attachments related to the report, such as the Pre-Project Plan and the Project Handbook.

1.3 Acronyms and Abbreviations

- **AST** - Abstract Syntax Tree
- **EEG** - Electroencephalogram
- **GUI** - Graphical User Interface
- **IR** - Information Retrieval
- **LDA** - Latent Dirichlet Allocation
- **LSI** - Latent Semantic Indexing
- **NTNU** - Norges teknisk-naturvitenskapelige universitet
- **OOP** - Object-Oriented Programming
- **PCA** - Principal Component Analysis
- **STD** - Standard Deviation
- **SVD** - Singular Value Decomposition

2 Theory

2.1 Literature review

In this section, we review a number of articles that pertain to the differences between professional and novice programmers, as well as the use of AST to represent code. The objective of this review, is to highlight the differences between professionals and novices, thus supporting our underlying hypothesis, that the analysis of professional source code can serve in strengthening the education of beginner programmers.

***Programming Pedagogy – A Psychological Overview* (Winslow)**

The ability to solve problems serves as a significant indicator of a programmer's competence level. Expert programmers showcase their proficiency by utilizing a diverse range of schemata and patterns to categorize problems, a skill that novices often lack (Windslow, 1996). Novices face challenges in understanding "where and how to combine statements to generate the desired result, (Windslow, 1996)" in contrast to experts. As beginners possess superficial knowledge, they need to acquire fundamental facts and problem-solving skills to enhance their programming abilities, and master the field. To support this learning process and enable progression towards higher proficiency, models play a crucial role. In this context, a model refers to a cognitive framework or structure used to organize knowledge, approach problem-solving tasks, and understand concepts (Windslow, 1996). A model might include problem-solving strategies and knowledge of programming syntax. As suggested by Linn and Clancy (1992), employing techniques such as verbal descriptions, illustrations, and connections to other concepts can aid in the development of these models. By utilizing such models, beginners can strengthen their understanding and advance their programming expertise (Windslow, 1996).

***Program Understanding – A Survey* (von Mayrhauser and Vans)**

The ability of programmers to understand code is greatly influenced by their level of expertise. According to Mayrhauser and Vans (1994), experts in programming develop specialized schemas through experience, enabling them to recognize familiar patterns in code and bypass in-depth analysis. These schemas play a vital role in code understanding and facilitate problem-solving. However, the comprehension of unconventional code poses challenges even for experts, as it often lacks suitable schemas that can be readily applied. Consequently, both novice and expert programmers may struggle to comprehend unconventional programming styles and algorithms due to the absence of matching schemas in their long-term memory (Mayrhauser and Vans, 1994).

***Comparing Programming Language Comprehension between Novice and Expert Programmers using EEG Analysis* (Lee et al.)**

In a study by Lee et al. (2016), novice and expert programmers were compared in terms of programming language comprehension using electroencephalogram (EEG) analysis. While both groups performed similarly in terms of correctness on a relatively easy programming task, the experts exhibited a significant advantage in response time (Lee et al., 2016). This suggests that experts are not necessarily more accurate than novices in simple tasks, but they excel in efficiently solving programming problems.

The EEG analysis provided insights into the neural mechanisms underlying programming comprehension. The expert group showed higher power in the Beta range, indicating a greater utilization of cognitive skills such as logical thinking and conscious thought. Moreover, the experts demonstrated higher activation of the right hemisphere, which is associated with understanding abstract ideas and processing new information (Lee et al., 2016). These findings suggest that experts employ advanced cognitive processes and have a better grasp of complex programming concepts, leading to their superior performance and efficiency in programming tasks.

***Programming problem representation in novice and expert programmers* (Weiser and Shertz)**

The research paper examines the representation of computer programming problems in relation to the organization of programming knowledge (Weiser and Shertz, 1983). The study replicates an experiment previously done for physics knowledge (Chi, Feltovich, and Glaser, 1981) to examine differences in the categories used for problem representation by novice and expert programmers. Results from sorting tasks show that experts and novices begin their problem representations with specific different problem categories. Experts tend to start with a more abstract approach, focusing on the underlying algorithmic structure of the problem. In contrast, novices tend to focus more on the surface features of the problem, such as specific data inputs and outputs. The experts' analysis reveals a consistent categorization of programming problems, which differs from the categorization by novice programmers. Furthermore, the study concludes that novice programmers are not consistent in their labeling of programming problems (Weiser and Shertz, 1983).

***A Novel Neutral Source Code Representation Based on Abstract Syntax Tree* (Zhang et al.)**

AST is a type of tree structure that represents the abstract syntactic structure of source code (Baxter, 2016). ASTs are widely used by programming and software engineering tools to describe the lexical information and the syntactic structure of source code, such as method names and control flow structures. Compared with plain source code, ASTs are abstract and do not include all details such as punctuation and delimiters (Zhang et al., 2019).

Traditional approaches such as Information Retrieval (IR) (T. Kamiya, Kusumoto, and Inoue, 2002), Latent Semantic Indexing (LSI) (Deerwester et al., 1990) and Latent Dirichlet Allocation (LDA) (Blei, Ng, and Jordan, 2003) usually treat code fragments as natural language text and model them based on tokens (Zhang et al., 2019). According to Panichella et al. (2013), the common problem of these approaches is that they assume the underlying corpus (i.e. the source code) is composed of natural language text (Zhang et al., 2019). Even though code fragments have something in common with plain text, they should not be simply dealt with text-based or token-based methods due to their richer and more explicit structural information (Baxter, 2016; Mou et al., 2016).

***Heterogeneous tree structure classification to label Java programmers according to their expertise level* (Ortin et al.)**

The research paper utelized AST for feature representation of Java source code to construct a classification model. The source code database consisted of 35,309 Java files, and got represented by a dataset containing 12.5 million AST nodes. The research results showed that the system was able to label expert programs (from open-source projects) and novice programs (from student assignments) with an accuracy of 99.6%. Additionaly, it seemed to be easier for the model to identify the syntax patterns written by experts, rather than by novices (Ortin et al., 2020).

2.2 Principal Component Analysis (PCA)

PCA is a statistical technique used to analyze the dimensionality of datasets (Jolliffe, 1986; Strang, 2009). The primary objective of PCA is to reduce the dimensionality and size of the dataset, while still preserving the indispensable information that constitutes the majority of the dataset. This is accomplished by creating new variables known as principal components, which are linear combinations of the original variables. The first principal component aims to capture the maximum variance in the dataset. Each subsequent principal component, captures the variance in the space orthogonal to the previous principal components. All principal components are orthogonal, providing distinct and uncorrelated information about the dataset. Mathematically, PCA relies on singular value decomposition (SVD) for the analysis of rectangular matrices. SVD is a technique that decomposes a matrix $\boldsymbol{D}$ into three simpler matrices (Abdi and Williams, 2010). The mathematical representation of SVD is given by:

$$D = U\Sigma V^T$$

The columns of the orthogonal matrices U and V^T are called the left and right singular vectors, respectively. The singular values, stored in the diagonal matrix Σ, indicate the significance of each principal component in explaining the variance in the data(Abdi and Williams, 2010). The explained variance ratio for

a specific principal component, denoted by the index i, is calculated as follows:

$$Explained\ Variance\ Ratio_i = \frac{\sigma_i^2}{\sum_{k=1}^{n} \sigma_k^2}$$

In this equation, σ represents a singular value from the matrix Σ , and n is the total number of singular values. The numerator is the square of the singular value corresponding to the principal value in question, while the denominator is the sum of the squares of all singular values. The cumulative explained variance ratio is utilized to determine the required number of principal components to achieve a desired ratio of explained variance (Abdi and Williams, 2010).

2.3 *k*-means Clustering

k-means is a popular clustering algorithm, utilized for unsupervised machine learning. The goal of the algorithm is to divide the data points of a dataset into a predetermined number of clusters, k, where each cluster contains data points that are close to the mean of the cluster in an Euclidean sense (Han, Pei, and Kamber, 2011).

2.4 Silhouette Score

The silhouette score is a metric used to evaluate the quality of clustering results. It assesses both the compactness and separation of clusters, and provides a value ranging from -1 to 1. A high score indicates well-defined and compact clusters, while a low score suggests overlapping or ambiguous clusters. Scores close to 0 indicate the presence of clusters that overlap or are not clearly separated. The silhouette score for an individual data point is computed using the following formula(Januzaj, Beqiri, and Luma, 2023):

$$s = \frac{b - a}{max(a, b)}$$

In this equation, a represents the average distance between the given point and other data points within the same cluster, while b represents the average distance between the data point and the data points in the nearest neighboring cluster. To obtain the silhouette score for the clustering result as a whole, the individual silhouette scores of all data points are averaged (Januzaj, Beqiri, and Luma, 2023).

3 Method

3.1 Research Method

Research methodology refers to systematic and structured approaches used in scientific research to collect, analyze, and interpret data. It includes quantitative and qualitative methods. Quantitative research focuses on numerical data, using statistical analysis to quantify phenomena and relationships. Qualitative research, on the other hand, seeks to understand complex phenomena in their natural context through words, images, and descriptions. Researchers can also use a combination of both methods, to gain a comprehensive understanding of a research question.

For this research project, we undertook a quantitative analysis of source code from 30 open-source libraries to identify qualities of professional Python code. To establish a solid foundation for our study, we conducted a literature review and formulated three research questions, as detailed in Section 1.1. By utilizing the Python module AST, the source code was transformed into a numerical matrix representation, which is a suitable format for analysis (Zhang et al., 2019). To analyze the data, we utilized various statistical techniques including PCA, k-means clustering, the silhouette score, and descriptive statistics. The analysis was structured based on two Python textbooks, and the findings regarding the qualities of professional source code are presented in Section 4. Finally, we drew conclusions to the research questions in Section 5. The illustration below depicts the methodology employed in the research project.

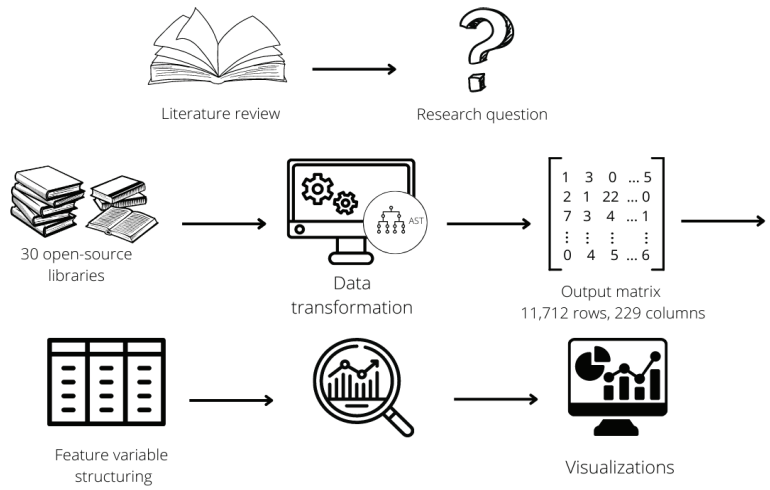


Figure 1: Illustration of applied research methods.

3.2 Abstract Syntax Tree (AST)

AST is a data structure used in programming to represent the structure of source code (Python Software Foundation, 2021a). It serves as an intermediate representation between the original source code and the eventual machine code or execution. The AST represents the code in a tree-like structure, where each node corresponds to a specific syntactic element, such as variables, *functions*, or *if* statements. The representation is abstract and do not include details such as comments, punctuation and delimiters (Zhang et al., 2019). The AST helps in capturing the relationships and hierarchy between these elements, enabling various operations like analysis, optimization, and transformation of the code (Kluyver, n.d.).

To analyze the characteristics of the dataset, which comprises 30 open-source Python libraries encompassing a total of 11,712 files, we utilized a program that leverages AST to extract information about the nodes. The extracted data was then quantified for subsequent analysis.

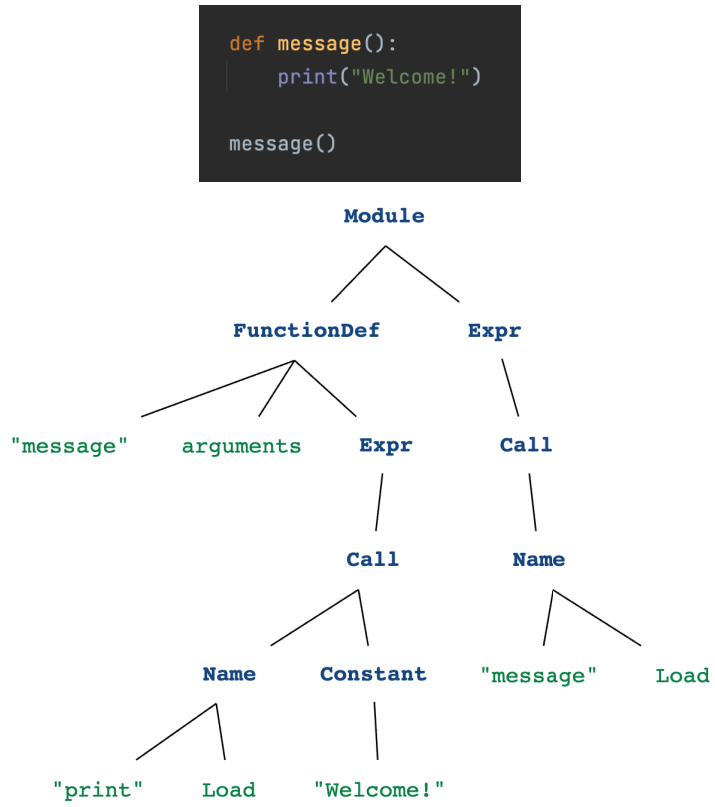


Figure 2: Example of AST visualisation. Code snippet and the corresponding AST tree visualization.

3.3 Libraries

In this study, expert code refers to source code written by expert developers. To satisfy the condition that the code is written by professionals, the project supervisor collected code from GitHub with certain requirements:

- The project had to be written in Python 3.
- The code had to be derived from mature projects that were recognised and measured by reports and feedback by a wide group of Python developers. Therefore, the project had at least 50 contributors.
- The project had to be updated within the past year.
- The project had to have more than 10,000 GitHub stars, which is a measure of whether the library is popular.

Based on the above requirements, 30 Python 3 libraries were collected, including: algorithms, Ansible, beets, Borg, Boto3, Cerberus, conda, cryptography, Django, Errbot, FastAPI, Gooey, Keras, Luigi, Matplotlib, mitmproxy, NLTK, Numba, pandas, pip, PlatformIO, psutil, pygame, Python Fire, PythonRobotics, Scapy, scikit-learn, Scrapy, sqlmap, and thumbor. See Table 1 for statistics of the libraries, and version numbers.

Table 1: Overview of the 30 utilized open-source libraries.

Nr.	Library Name	Version	Number of Files
1.	Ansible	2.12.1	1066
2.	Borg	1.1.17	54
3.	Boto3	1.18.1	81
4.	Cerberus	1.3.4	92
5.	conda	4.10.3	199
6.	cryptography	3.4.7	137
7.	Errbot	9.9.9	105
8.	Gooey	1.0.8.1	112
9.	Keras	2.7.0	392
10.	Luigi	1.14.2	220
11.	Matplotlib	3.4.2	752
12.	NLTK	3.6.2	255
13.	Numba	0.53.1	456
14.	pandas	1.3.0	1018
15.	pip	21.1.3	439
16.	PlatformIO	5.2.0	160
17.	psutil	5.8.1	53
18.	pygame	2.0.1	132
19.	Python Fire	0.4.0	49
20.	PythonRobotics	1.0.0	168
21.	Scapy	2.4.5	230
22.	scikit-learn	1.0	653
23.	sqlmap	1.5	396
24.	thumbor	7.0.0a5	170
25.	algorithms	1.0.1	378
26.	beets	1.5.0	148
27.	Django	4.0.0	1832
28.	FastAPI	0.65.1	945
29.	mitmproxy	7.0.0	735
30.	Scrapy	2.5.0	284

3.4 How We Decided What to Visualize

As mentioned earlier, the 11,712 files yielded 229 feature variables, offering numerous possibilities for analysis and visualization. These variables include counts of functions, classes, binary operations, conditions, and loops, among others. However, organizing such a high-dimensional feature space into lower-dimensional visualizations is challenging due to the intractable number of possible combinations.

After carefully examining the feature space and considering the relationship between the numerical values and the source code, we decided to focus on meaningful visualizations for first-year university programming students. To accomplish this, we conducted a reflective search to determine the types of visualizations that beginner programmers might find helpful in lectures.

To create lower-dimensional subspaces, we analysed the structures of introductory Python programming textbooks, specifically "Python for Everyone" by Horstmann and Necaise (2019) and "Starting Out with Python" by Gaddis (2019). We found that these two books, which are part of NTNU's educational programming syllabus, have similar structures. Table 2 displays the chapters covered in the examined textbooks and how we decided to structure the analysis.

Table 2: Visualisation structure, based on two programming textbooks, *Python for Everyone* Horstmann and Necaise, 2019, and *Starting Out with Python* Gaddis, 2019.

Chapter(s) in Python for Everyone	**Chapter(s) in Starting Out with Python**	**Our Chapter Structure**
1. Introduction	1. Introduction to Computers and Programming	1. Keywords
2. Programming with Numbers and Strings	2. Input, Processing, and Output 8.More About Strings	2. Variable Assignments and Augmentations
3. Decisions	3. Decision Structures and Boolean Logic	3. Decisions
4. Loops	4. Repetition Structures	4. Loops
5. Functions	5. Functions	5. Functions
6. Lists 8. Sets and Dictionaries	7. Lists and Tuples 9.Dictionaries and Sets	6. Data Structures
7. Files and Exceptions	6. Files and Expetions	7. Files and Exceptions
9. Objects and Classes	10. Classes and Object Oriented Programming	8. Classes
10. Inheritance	11. Inheritance	
11. Recursion	12. Recursion	
	13. GUI Programming	
12. Sorting and Searching		

Thus, our goal was to map the numerical features extracted from the source code to the appropriate chapter content. We excluded chapters related to inheritance, recursion, GUI programming, and sorting algorithms from our analysis.

Furthermore, to enable creative work and facilitate the development of visualizations, the feature variables were organized into tables based on their associated concepts. Within the tables, the variables were grouped by category. For most concepts, the categories were "General", "Content of Concept" and "Location of Concept in Code". The tables were in turn arranged according to the chosen chapter structure. Tables 3 - 13 shows the variables related to each section.

Table 3: Overview of variables related to Section 4.4 about variable assignments and augmentations.

General (4)	**Location in code (15)**
Assign_node	Assign_class
	Assign_for
	Assign_fun
	Assign_if
	Assign_while
AugAssign_node	AugAssign_class
	AugAssign_for
	AugAssign_fun
	AugAssign_if
	AugAssign_while
AnnAssign_node	AnnAssign_class
	AnnAssign_for
	AnnAssign_fun
	AnnAssign_if
	AnnAssign_while

4 Results

This chapter presents the results obtained by analyzing professional, open-source Python code. The results follow the analysis of the textbooks' contents, presented in Section 3.4.

A consistent trend in the results is that functions are the main building block of the analyzed source code. Furthermore, the average length of if statements, loops, and functions is found to be short, indicating that programmers prioritize modular code.

The average length is computed by dividing the total number of lines dedicated to each code block in a file by the corresponding count of occurrences. Files without any instances of a particular construct are not considered in this calculation. In this section, we consider the location of *if* statements, loops, *functions*, and *classes* within the code. The location in the code is divided into six groups, namely *functions*, *while* loops, *if* statements, *for* loops, *classes*, and *global*. Specifically, we explore whether a code block, such as a *function* definition, occurs within an *if* statement, a loop, a *class*, or the *global* scope.

Table 4: Overview of variables related to Section 4.5.1 about *if* statements.

General (6)	**Content of if statements (23)**	**Location of if statements (4 + 1)**
if	Expr_if	If_fun
elif	Assign_if	If_for
else	*If_if*	*If_if*
Nu_if	Return_if	If_while
Sum_len_if	For_if	If_class
IfExp_node	Raise_if	
	Break_if	
	Pass_if	
	AugAssign_if	
	Continue_if	
	With_if	
	Try_if	
	ImportFrom_if	
	While_if	
	Delete_if	
	Import_if	
	FunctionDef_if	
	ClassDef_if	
	Assert_if	
	AnnAssign_if	
	Global_if	
	Nonlocal_if	
	AsyncFunctionDef_if	

Table 5: Overview of variables related to Section 4.5.2 about comparison operator tokens.

General (1)	**Relational operators (6)**	**Identity and membership operators (6)**
Compare_node	Eq_node	Is_node
	NotEq_node	IsNot_node
	Lt_node	In_node
	LtE_node	NotIn_node
	Gt_node	is
	GtE_node	in

Table 6: Overview of variables related to Section 4.6.1 about *while* loops.

General (3)	**Content of while loops (14)**	**Location of while loops (3 + 1)**
while	Assign_while	While_fun
Nu_while	If_while	*While_while*
Sum_len_while	Expr_while	While_if
	Try_while	While_for
	While_while	
	AugAssign_while	
	For_while	
	Pass_while	
	Delete_while	
	Break_while	
	Assert_while	
	Return_while	
	With_while	
	AnnAssign_while	

Table 7: Overview of variables related to Section 4.6.2: about *for* loops.

General (3)	**Content of for loops (17)**	**Location of for loops (6 + 1)**
for	Assign_for	For_fun
Nu_for	If_for	For_if
Sum_len_for	Expr_for	*For_for*
	For_for	FormattedValue_node
	Try_for	For_while
	AugAssign_for	For_class
	With_for	AsyncFor_node
	While_for	
	FunctionDef_for	
	Delete_for	
	Assert_for	
	Raise_for	
	Return_for	
	Pass_for	
	ClassDef_for	
	Break_for	
	AnnAssign_for	

Table 8: Overview of variables related to Section 4.7 about *functions*.

General (6)	**Content of functions (21)**	**Location of function definitions (5 + 1)**
def	Expr_fun	FunctionDef_class
Nu_fun	If_fun	FunctionDef_if
Sum_FunName_fun	Assign_fun	*FunctionDef_for*
Sum_len_fun	Raise_fun	AsyncFunctionDef_node
Call_node	Return_fun	AsyncFunctionDef_class
lambda	For_fun	AsyncFunctionDef_if
Lambda_node	Delete_fun	
	Try_fun	
	While_fun	
	With_fun	
	Import_fun	
	FunctionDef_fun	
	ImportFrom_fun	
	AugAssign_fun	
	Pass_fun	
	Global_fun	
	Assert_fun	
	ClassDef_fun	
	Nonlocal_fun	
	AnnAssign_fun	
	AsyncFunctionDef_fun	

Table 9: Overview of variables related to Section 4.8 about data structures.

Data structures (4)	**Comperhension (3)**
List_node	ListComp_node
Tuple_node	SetComp_node
Set_node	DictComp_node
Dict_node	

Table 10: Overview of variables related to Section 4.9.1 about *raise*.

General (2)	**Location of raise (5)**
raise	Raise_fun
Raise_node	Raise_if
	Raise_for

Table 11: Overview of variables related to Section 4.9.2 about *try*.

General (2)	**Location of try (5)**
try	Try_fun
Try_node	Try_while
	Try_if
	Try_for
	Try_class

Table 12: Overview of variables related to Section 4.9.3 about *with*.

General (4)	**Location of with (4)**
with	With_fun
With_node	With_if
withitem_node	With_for
AsyncWith_node	With_while

Table 13: Overview of variables related to Section 4.10 about *classes*.

General (4)	**Content of class (12)**	**Location of class definitions (3 + 1)**
class	Expr_class	ClassDef_if
Nu_Classes	Assign_class	ClassDef_fun
Sum_ClassName_class	FunctionDef_class	*ClassDef_class*
Sum_len_class	Pass_class	ClassDef_for
	Try_class	
	ClassDef_class	
	If_class	
	AugAssign_class	
	AnnAssign_class	
	AsyncFunctionDef_class	
	Global_class	
	For_class	

4.1 Data Dimensionality

As mentioned in Section 3.2, the professional source code was transformed into numerical values using Python's AST. Each source code file was represented as a vector in a 229-dimensional space, with 176 features derived directly from the AST nodes. The remaining features represent 35 Python keyword frequencies, 15 length counts, and counts of the function calls *input*, *print*, and *open*.

In this section, we analyze the dimensionality of our data. Our goal is to determine if there exists a reduced space that captures all the variations in the dataset. To achieve this, we utilized PCA to transform the data into an optimal reduced space.

Figure 3 shows the plot of the singular values' ratio associated with the singular vectors. We observe that the first singular value accounts for 16% of the data, while it takes 181 dimensions to capture 99.9% of the data variation. Notably, this number is very close to the actual number of nodes in the AST. Furthermore, since the transformation of source code to the AST structure is a lossless operation, the additional features related to keywords and arguments in functions are shown to be mathematically unnecessary.

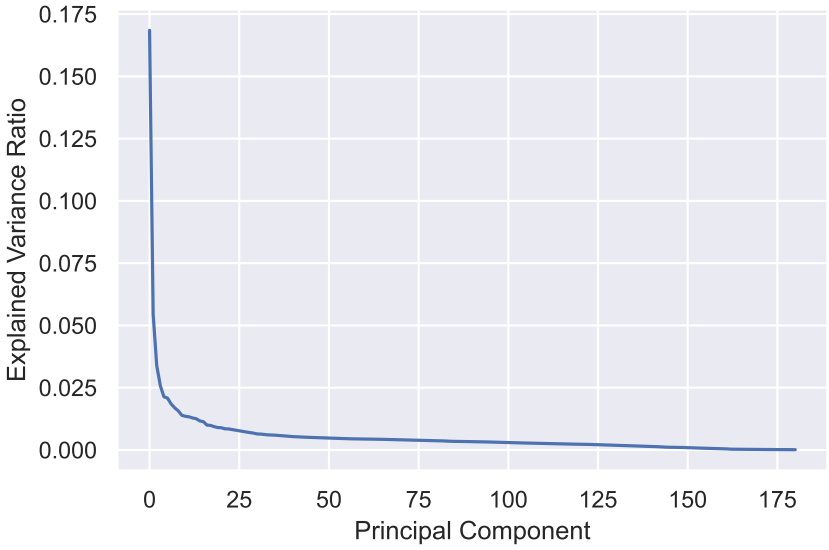


Figure 3: The explained variance ratio for each principal component up to 181. We note that the majority of the variance is explained by the first few principal components.

4.2 Clustering

Previously, we assumed that all code within the 30 Python 3 libraries we examined could be considered as one cohesive unit of professional source code. However, we have not verified this assumption.

To investigate the similarity of the professional code in our dataset, we utilized the k-means clustering algorithm. Our hypothesis was that professional code from different libraries would not form distinct clusters. If clustering results showed otherwise, our hypothesis would require adjustment, and individual clusters would need separate analysis.

Figure 4 depicts the silhouette score obtained from k-means. Notably, the highest score occurred when the data were divided into a maximum of two clusters. These results support the hypothesis that professional source code in our dataset can be treated as a single cohesive unit. Finally, Figure 5 presents the two-dimensional projection of the data, showcasing the corresponding clusters. Detailed analysis of the differences between the two clusters are considered future work.

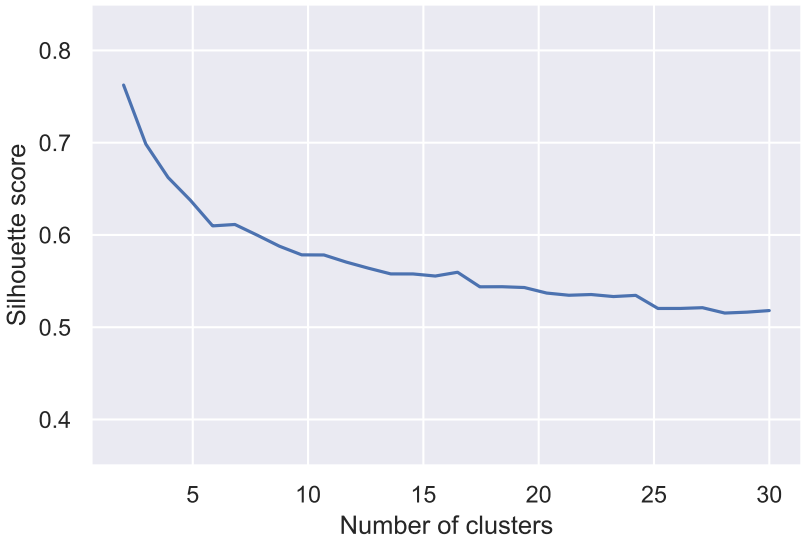


Figure 4: The silhouette score for different number of clusters, ranging from 2 to 30. The highest score is obtained using 2 clusters, signifying the most suitable number of clusters for our dataset.

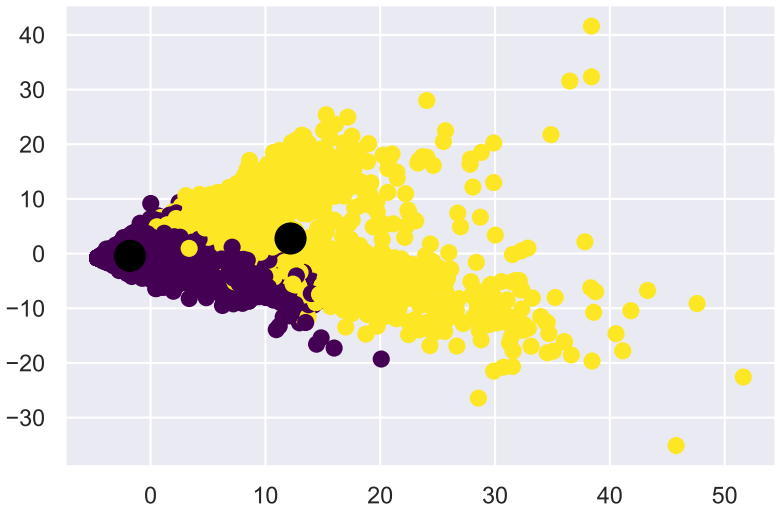


Figure 5: The two clusters in the dataset, represented in a two-dimensional projection.

4.3 Keywords

This section, presents an analysis of the proportion of the 35 Python 3 keywords across all libraries, as well as exploring the distribution of keywords within each library.

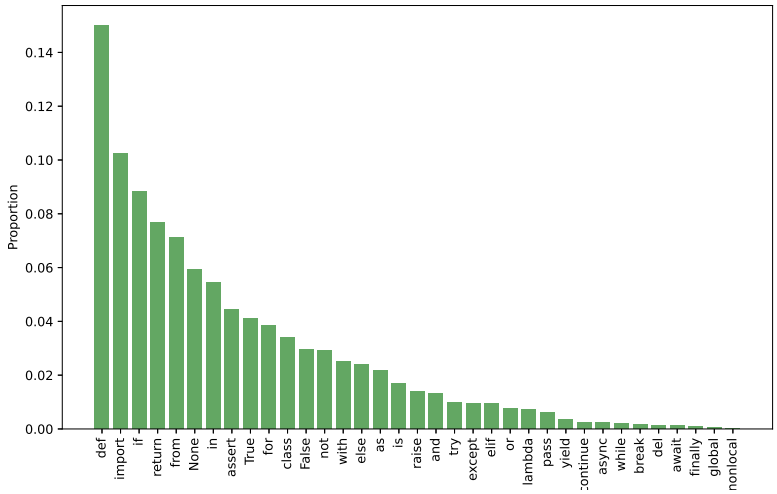


Figure 6: The most frequently used keywords are: *def, import*, and *if*. The least frequently used keywords are: *nonlocal, global*, and *finally*.

Figure 6 presents the proportional distribution of keywords across 30 libraries, highlighting that the most frequently used keywords are *def, import, if, return*, and *from*. When analyzing the individual libraries, we observed variation in the use of the keywords across the libraries. Our analysis reveals the following patterns: the keywords *def*, *if*, and *return* consistently rank among the top ten most commonly used keywords. The keyword *import* appears in the top ten for all libraries except one, algorithms (library 25), while the keywords *None* and *in* are among top ten in all libraries except two.

Our analysis shows that the keywords *nonlocal*, *global*, *finally*, *await*, and *del* are consistently among the least utilized keywords across 28 of the 30 libraries studied. However, thumbor (library 24) and FastAPI (library 28) exhibit notable differences, with *await* being the 12th and 14th most used keyword, respectively. Moreover, it is worth mentioning that *nonlocal* is only employed in two libraries, specifically Algorithms (library 25) and Mitmproxy (library 29), indicating its limited adoption within the analyzed source code. Similarly, *await* and *async* are used in only six libraries, namely Platformio (library 16), Thumbor (library 24), Django (library)27, FastAPI (library 28), Mitmproxy (library 29), and Scrapy (library 30). Conversely, *global* is utilized in half of the libraries.

Further analysis is required to clearly understand the correlations between the keywords. However, we can already see that keywords are highly correlated. For example, function definitions (*def*) are correlated with *return* and *None*.

Finally, Table 14 presents the top seven most frequently used keywords in each library. Notably, either *def* or *if* consistently emerge as the most commonly used keywords across all libraries, indicating that functions and conditions are the predominant building blocks in the source code.

Table 14: Top seven most utilized keywords.

Library		**Top seven most utilized keywords**						
All libraries combined		def	import	if	return	from	None	in
1.	Ansible	if	def	return	in	import	None	not
2.	Borg	if	def	return	in	import	for	None
3.	Boto3	def	return	if	import	in	from	for
4.	Cerberus	def	True	None	if	import	from	assert
5.	conda	if	import	def	return	from	in	for
6.	cryptography	def	if	return	import	raise	not	from
7.	Errbot	def	if	return	None	in	import	not
8.	Gooey	def	return	if	None	import	in	for
9.	Keras	if	def	None	return	import	not	from
10.	Luigi	def	if	return	import	None	in	for
11.	Matplotlib	def	if	None	return	for	import	assert
12.	NLTK	if	def	return	in	for	None	import
13.	Numba	def	return	if	in	import	None	for
14.	pandas	def	if	None	return	import	assert	in
15.	pip	if	def	return	None	import	not	from
16.	PlatformIO	if	return	def	in	import	not	from
17.	psutil	def	if	import	in	return	from	for
18.	pygame	if	def	import	True	in	return	for
19.	Python Fire	if	return	None	import	def	in	else
20.	PythonRobotics	if	in	def	for	return	None	True
21.	Scapy	if	return	def	class	None	import	from
22.	scikit-learn	if	def	import	from	None	return	in
23.	sqlmap	if	import	from	def	return	not	in
24.	thumbor	if	None	return	def	import	not	from
25.	algorithms	if	return	def	in	for	and	None
26.	beets	if	return	def	in	for	import	not
27.	Django	def	if	return	None	import	in	from
28.	FastAPI	if	import	None	return	def	in	and
29.	mitmproxy	if	def	return	import	None	in	from
30.	Scrapy	def	if	return	import	from	None	in

4.4 Variable Assignments and Augmentations

In this section, we present the results of our analysis on the relationship between assignments and augmentations. In out data, assignments are represented by the "Assign" node in the AST, which corresponds to variable assignment statements like "x = 5". On the other hand, augmentations are represented by the "AugAssign" node in the AST, which denotes augmented assignment statements involving a binary operator combined with the assignment operator to update the variable in-place, such as "x += 1". It is important to note that the term "augmentation" typically implies an increase in values, but the AugAssign node is not limited to only incrementing the variable value. It can perform other operations based on the chosen binary operator.

Figure 7 illustrates the correlation between assignments and augmentations. The Pearson's correlation coefficient is 0.219, indication a low correlation between assignments and augmentations. It is worth noting that not all assignments allows for augmentations, and a correlation coefficient of 1 is therefore unlikely.

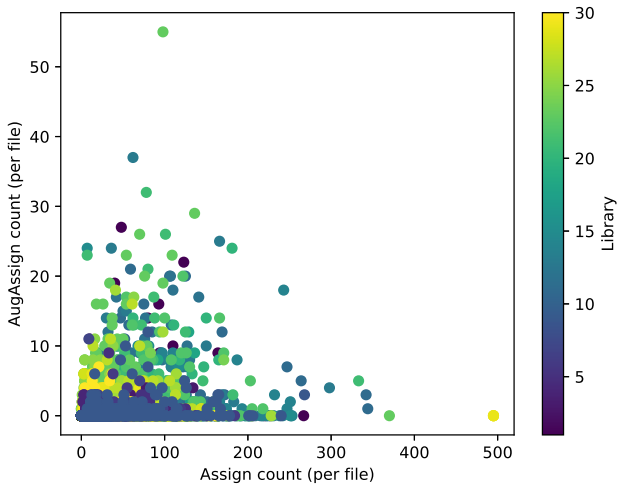


Figure 7: The correlation between variable assignment statements and variable augmentation statements. Each data point representing a file, and the hue axis represents which library the file belongs to.

4.5 Decisions

The analysis of decisions in source code includes the use of *if* statement, and comparison operators.

4.5.1 If Statements

Referring back to Figure 6, we observe that the keyword *if* is one of the most frequently used keywords in the analyzed dataset. Further, we note that the use of *else* and *elif* is less frequent, indicating that most of the *if* conditions are short, and simple. In other words, *if* blocks that don't include *elif* and *else* are the most common conditions.

To further analyze the use of *if* statements, we present our findings regarding the average number of lines within an *if* statement block per file, along with their placement within the code. Figure 8, depicts the distribution of the number of lines in an *if* statement block. We note that the distribution is positively skewed, indicating that the majority of the *if* statements have a low average length. To evaluate the data numerically, we calculated the average, median, standard deviation, minimum, and maximum. The results of the calculations are presented in Table 15, where we note that the mean value is 5, the standard deviation is 4, and the median is 4 lines per *if* statement. The values range from 1 to 127, with 93% of files having an average of 9 lines or fewer. This is derived from the mean plus one standard deviation, which is 9. Finally, we note that for the purpose of visualization, the data plotted in Figure 8 exclude the outliers.

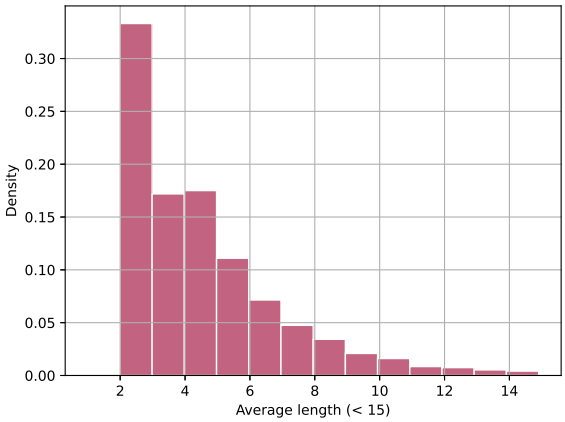


Figure 8: Average length of *if* statement per file. We note that for the purpose of visualization, the data outliers have been truncated. The majority of *if* statements are relatively short, with 93% of the files having an average length of 9 or less.

Table 15: Statistical summary of the average number of lines in *if* statements per file.

Mean	**Std**	**Median**	**Min**	**Max**
5	4	4	1	127

Figure 9 indicate that functions contain the majority of *if* statements, followed by *global* scope, *for* loops, nested *if* statements, and *while* loops. Figure 9 illustrates the distribution of *if* statements across all analyzed libraries. To assess whether the general distribution in Figure 9 is consistent among various libraries, we analyzed the probability of *if* statements per block in each library.

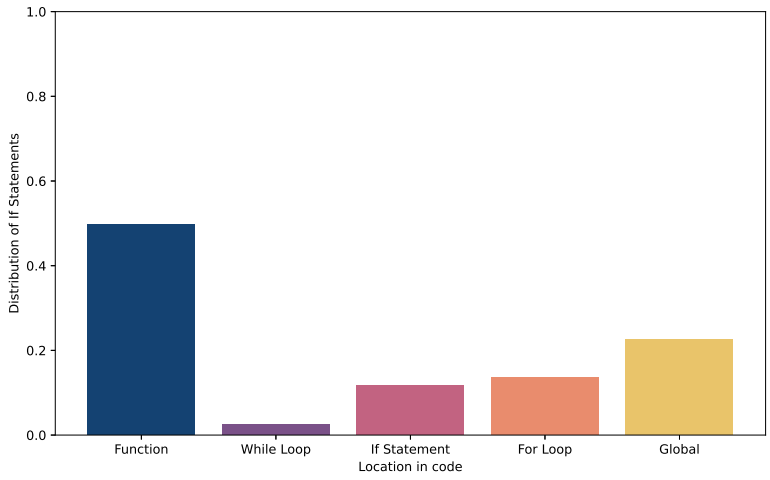


Figure 9: The probability distribution of *if* statement by location in code, across all libraries. The exact probability for each block are: *functions* 0.5, *while* loops 0.03, *if* statements 0.12, *for* loops 0.14, and *global* 0.22.

Figure 10 display the per-library distributions, which shows that the general probability distribution of *if* statements holds true across the majority of libraries. However, there are a few exceptions worth noting when examining the libraries individually. In the case of pygame (library 18), a larger proportion of *if* statements falls within *global* scope, accompanied by a lower proportion within *functions*. The distribution of *if* statements in PythonRobotics (library 20) and algorithms (library 26) deviates slightly from the overall trend, with a larger proportion within *while* loops. In these libraries, the order of prevalence, from highest to lowest, is as follows: *functions*, *for* loops, *global*, *while* loops, and *if* statements.

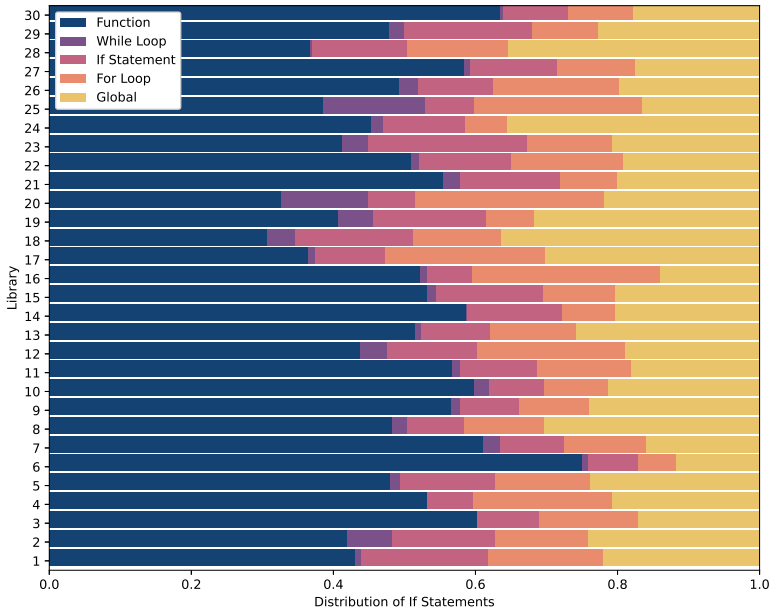


Figure 10: The probability distribution of *if* statements by location in code, per library. The majority of *if* statements are found within *functions*, while a small number are found within *while* loops.

4.5.2 Comparison operator tokens

In this section, we studied the utilization of comparison operator tokens. Specifically, the operators *==, !=, <, <=, >, >=, is, is not, in,* and *not in*. Figure 11 highlights the average number of each operator per file. Specifically, we calculate the values

$$\frac{\sum_{i=1}^{n} Op_i^j}{m}$$

where j stands for the type of operator, n stands for the number of operators of type j in a file, and m is the total number of files.

From Figure 11, we note that the comparison operator, *==*, is the most prevalent, followed by the operators *in*, and *is*. Further, we note that *==*, *in*, and *is*, are utilized more frequently than their negative counterparts *!=*, *not in*, and *is not*. Interestingly, the data reveal that the operator (*is*) is used almost twice as much as *is not*. Similarly, the operator *in* is used twice as much as *not in*. Specifically, the average count of positive comparison operator tokens is approximately twice as large as the count of negative comparison operator tokens. Further analysis of the use of operators and cases where the negative counterparts of the positive operators, such as *in* and *not in* are used, is left as considered as future work.

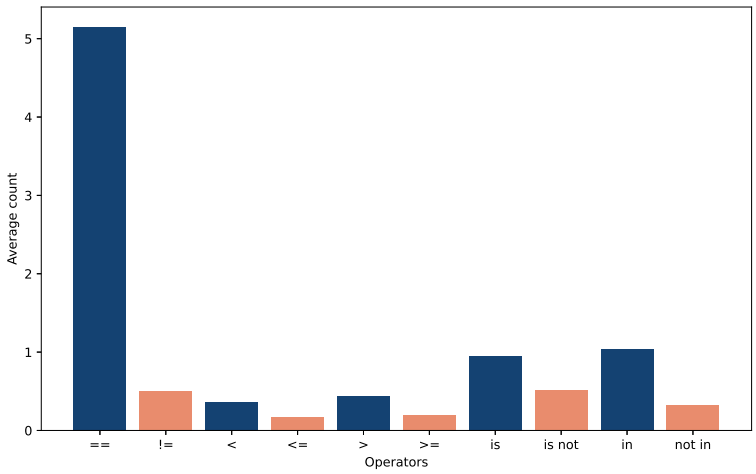


Figure 11: The average count of comparison operator tokens per file. The operators are as follows: *Equal ==*, *not equal !=*, *less than <*, *less than or equal to <=*, *greater than >*, *greater than or equal to >=*, *is*, *is not*, *in*, and *not in*.

4.6 Loops

This section, introduces the results of our analysis concerning the average line count of loops, as well as the expected location of the loops in the code. By loops, we mean both *while* loops and *for* loops.

4.6.1 While Loops

Figure 12 shows the distribution of the average length of *while* loops across all files. We note that the distribution is positively skewed, indicating that the majority of the *while* loops have a low average length. Specifically, 90% of the files have an average length of 28 lines or less, which is the mean plus one standard deviation. Table 16, lists the numerical values of the mean, standard deviation, median, minimum, and maximum average number of lines in a *while* loop per file. Finally, we note that for clarity of visualization, the data plotted in Figure 12 do not include the outliers.

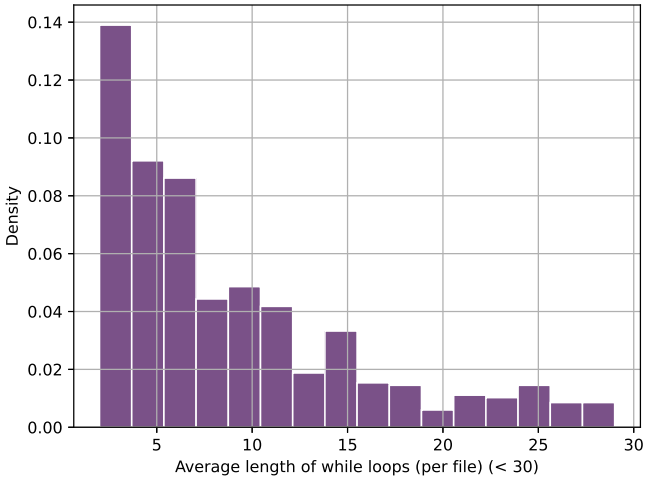


Figure 12: Average length of *while* loops per file. We note that for the purpose of visualization, the data outliers have been truncated. The majority of *while* loops are relatively short, with 90% of the files having an average length of 28 or less.

Table 16: Statistical summary of the average number of lines in *while loops* per file.

Mean	**Std**	**Median**	**Min**	**Max**
12	16	8	2	245

Figure 13 illustrates the probability distribution of the location of *while* loops across different code structures. It shows that the majority of *while* loops are found within *functions*, followed by *global* scope, *if* statements, *for* loops, and finally, nested *while* loops.

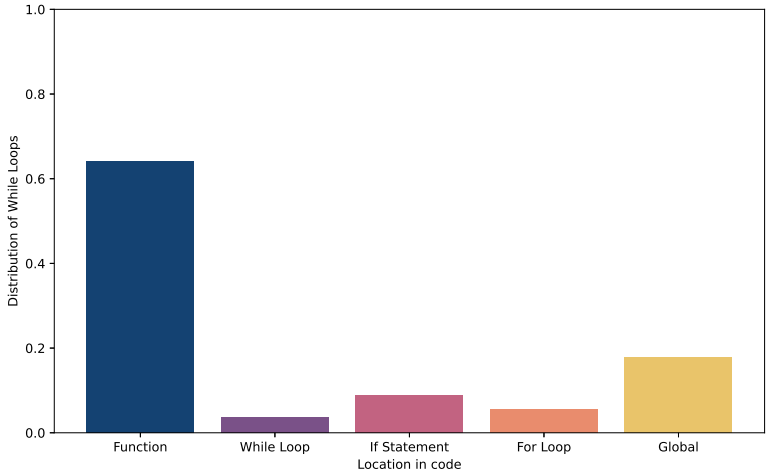


Figure 13: The probability distribution of *while* loop location, across all libraries. The exact probability distribution are as follows: *functions* 0.64, *global* 0.18, *if* statements 0.09, *for* loops 0.06, and *while* loops 0.04.

Figure 14 illustrates the distribution of *while loops* per library. The analysis reveals that in 80% of the analyzed libraries, *while loops* are predominantly located within *functions*. However, as shown in Figure 14, boto3 (library 3) stands out as an exception, displaying an even distribution of *while loops* within *if statements* and *for loops*. Three libraries, namely Gooey (library 8), psutil (library 17), and FastAPI (library 28), exhibit a higher distribution of *while* loops within the *global* scope compared to *functions*. Additionally, it is noteworthy that Cerberus (library 4) does not contain any *while loops*.

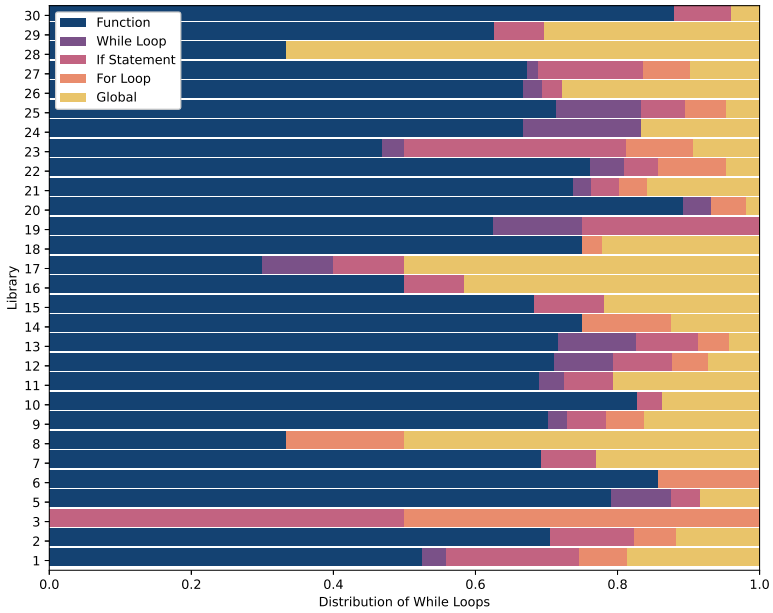


Figure 14: The probability distribution of *while* loops by location in code, per library. The majority of libraries have the highest proportion of *while* loops within *functions*.

4.6.2 For Loops

Figure 15 depicts the average distribution of the number of lines in an *for* loop. We note that the distribution is positively skewed, with the majority of the data lying to the left of the distribution. To evaluate the data numerically, we calculated the average, median, standard deviation, minimum, and maximum. The results of the calculations are tabulated in Table 17. We observe that the mean value is 7, the standard deviation is 7, and the median is 5 lines per *for* loop. The presence of outliers is indicated by the significant deviation of these values from the maximum value of 102. Specifically, 89% of the files have a average line count of 14 or less, which is the mean plus one standard deviation. Finally, it is important to note that for the purpose of visualization, the outliers have been excluded from the data plotted in Figure 15.

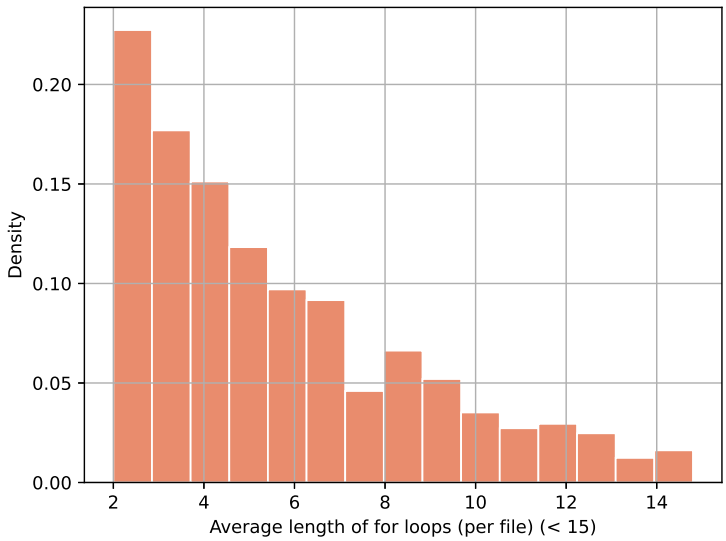


Figure 15: Average length of *for* loops per file. We note that for the purpose of visualization, the data outliers have been truncated. The majority of *for* loops are relatively short, with 89% of the files having an average length of 14 or less.

Table 17: Statistical summary of the average number of lines in *for* loops per file.

Mean	**Std**	**Median**	**Min**	**Max**
7	7	5	2	102

Figure 16 depicts the probability distribution of *for* loops in all libraries. *Functions* contain the majority of *for* loops, followed by *if* statements, nested *for* loops, *global*, and *while* loops. To assess the consistency of this general distribution across different libraries, we analyzed the probability of *for* loops per block in each library. Figure 17 display the per-library distributions.

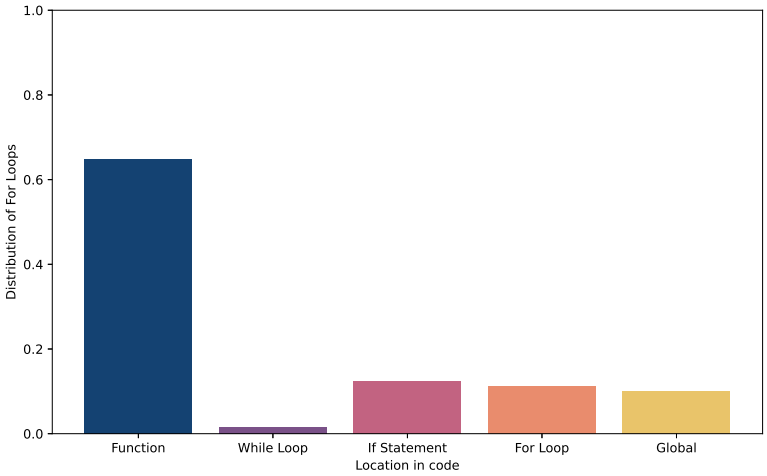


Figure 16: The probability distribution of *for* loops by location in code, across all libraries. The exact probabilities for each block are: *functions* 0.65, *while* loops 0.02, *if* statements 0.12, *for* loops 0.11, and *global* 0.10.

The overall trend in Figure 17 indicates that *for* loops are most commonly located within functions. Conversely, they are rarely found within while loops, with a zero probability observed in 50% of the libraries. However, it is worth noting that there are a higher occurrences of while loops in pygame (library 18).

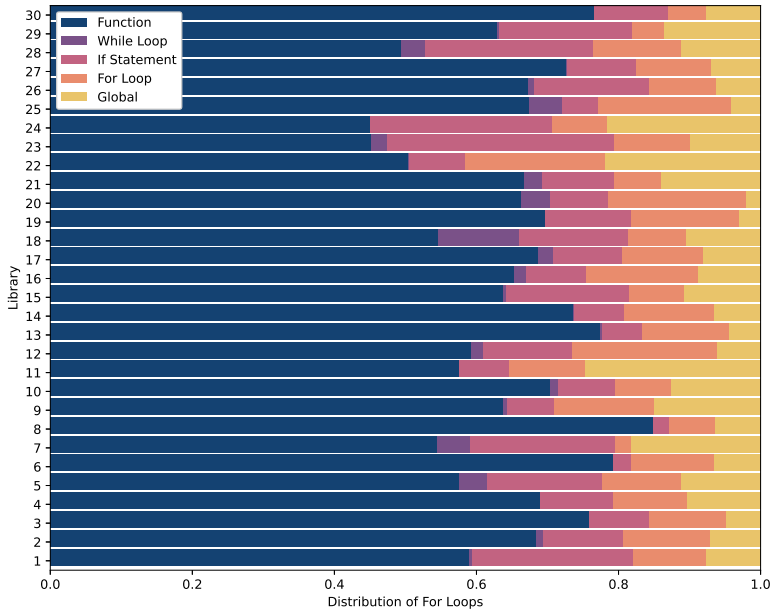


Figure 17: The probability distribution of *for* loops by location in code, across all libraries. *Functions* are the overall most likely location of *for* loops.

4.7 Functions

This section, introduces the results of our analysis of the average number of lines in a *function* per file, as well as the likely location of *functions* definitions in the code blocks.

Figure 18 presents the histogram of the average *function* length. We note that the data are positively skewed towards shorter average length. To further analyze the data, we calculated the average, median, standard deviation, minimum, and maximum average function lengths, shown in Table 18. Notably, the line counts range from 1 to 295, with 92% of *functions* having 28 or fewer lines, which corresponds to the mean plus one standard deviation. These data indicates that the majority of *functions* are relatively short. We note that, for visual representation purposes, the data plotted in Figure 18 exclude the few large outliers.

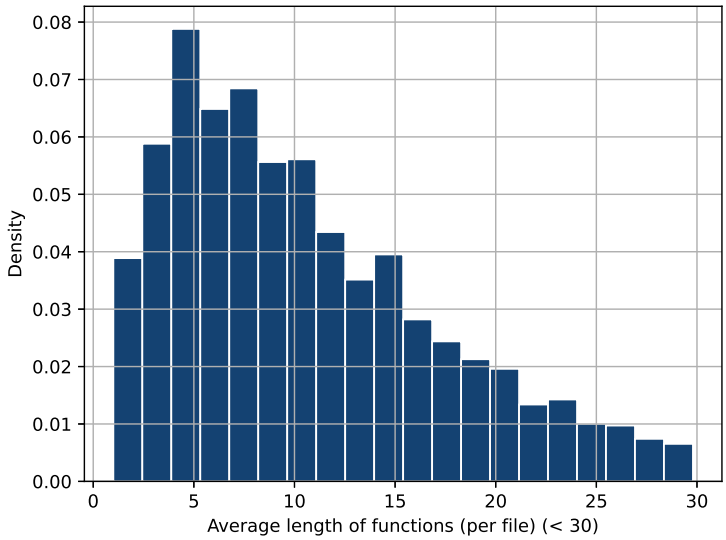


Figure 18: Average length of *functions* per file. We note that for the purpose of visualization, the data outliers have been truncated. 92% of files have a average length of 28 lines of less.

Table 18: Statistical summary of the average number of lines in *functions* per file.

Mean	**Std**	**Median**	**Min**	**Max**
14	14	10	1	295

Figure 19 illustrates the probability distribution of *function* definitions across the different code blocks. We observe that *functions* are predominantly located within *classes* (methods), followed by *global* scope, nested *functions*, and *if* statements. It is worth noting that there are only 26 cases of *functions* inside *for loops*, and no *functions* are found inside *while* loops. The examination of the very few cases were *functions* are defined inside *for* loops, is considered in future work.

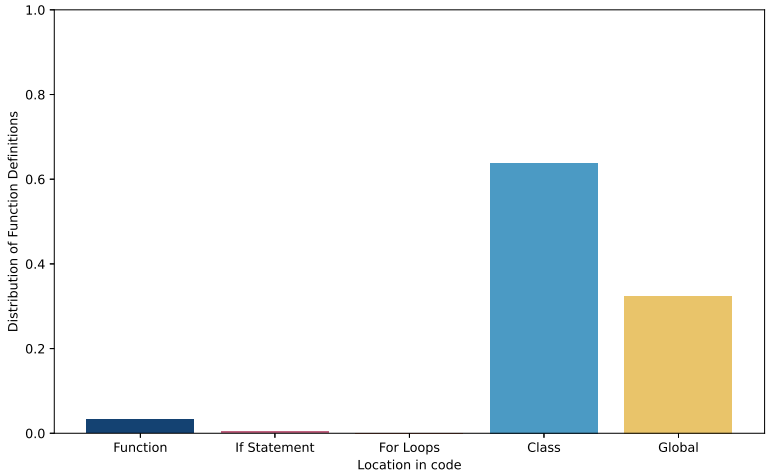


Figure 19: The probability distribution of *functions* by location in code across all libraries are as follows: *class* 0.64, *global* 0.32, *functions* 0.03, *if* statements 0.004, and *for* loops 0.0002.

Our analysis reveals a consistent trend across 83% of the libraries, highlighting that *functions* are primarily defined within *classes*. Notably, the libraries Cerberus (library 4), PythonRobotics (library 20), scikit (library 22), and FastAPI (library 28) have a higher likelihood of defining *functions* within the *global* scope compared to *classes*. Additionally, the library Conda (library 5) showcases an equal proportion of *function* definitions within both *classes* and the *global* scope. Numba (library 13) stands out with a notably higher proportion of nested *functions*. Furthermore, the occurrence of *function* definitions within *if* statements is relatively rare, as only seven libraries exhibit a nonzero probability. Cases were functions are defined within *if* statements are seen as future work.

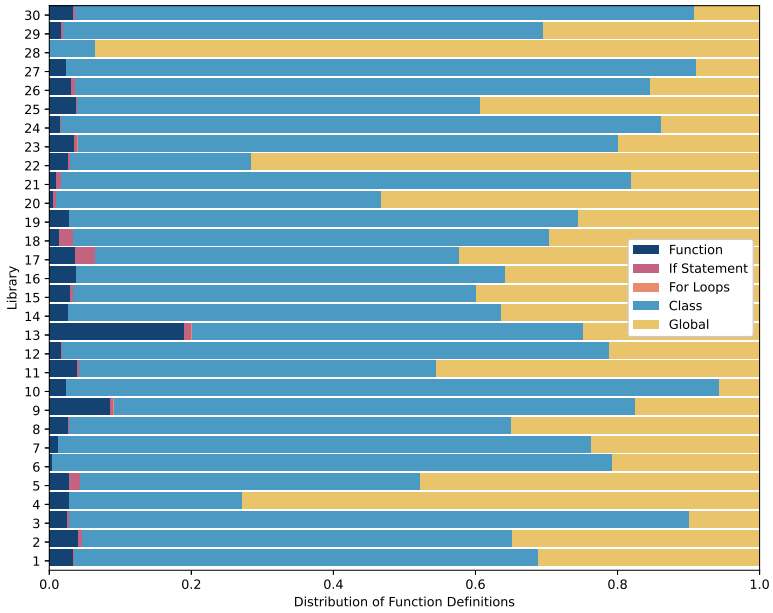


Figure 20: The probability distribution of *functions* by location in the code across each library. *Functions* are most likely located within *classes*, followed by the *global* scope.

4.8 Data structures

This section, presents the findings from our analysis of data structures. Specifically, four data structures were considered: *lists*, *tuples*, *sets* and *dictionaries*.

Figure 21 shows the distribution of data structures across the libraries. We note that *lists* and *tuples* are about equally popular, with a probability of 0.42 and 0.39, respectively. *Dictionaries* are also widely used, with a share of 0.19, while *sets* have the lowest rate at 0.0033.

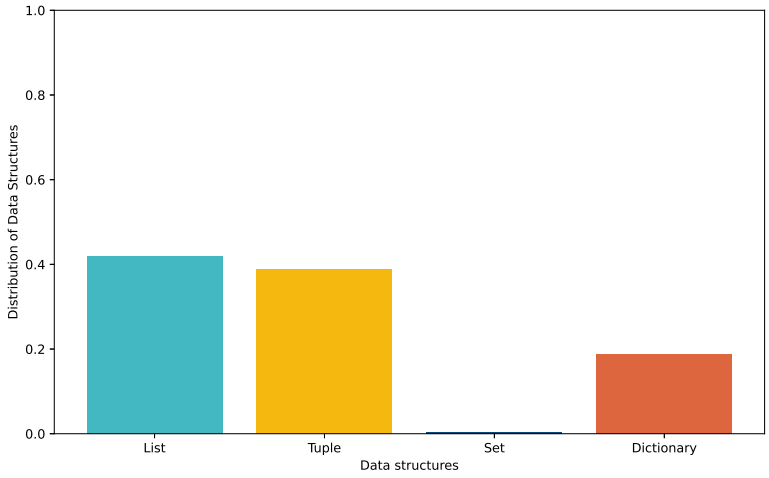


Figure 21: The probability distribution of data structures for all libraries combined. *Sets* are rarely used compared to the other considered data structures.

When examining the distributions per library in Figure 22, it becomes evident that the *set* data structure is rarely utilized in the analyzed libraries. In all libraries, the share of *sets* does not exceed 0.0015. Furthermore, it is noteworthy that *sets* are absent in eight libraries. In contrast, we note that *lists*, *tuples*, and *dictionaries* are utilized in every single library.

By enlarge, the examined libraries follow similar trends to that observed in Figure 21. The largest exceptions are found in Boto3 (library 3), Cerberus (library 4) and FastAPI (library 28), where *dictionaries* dominate the distribution.

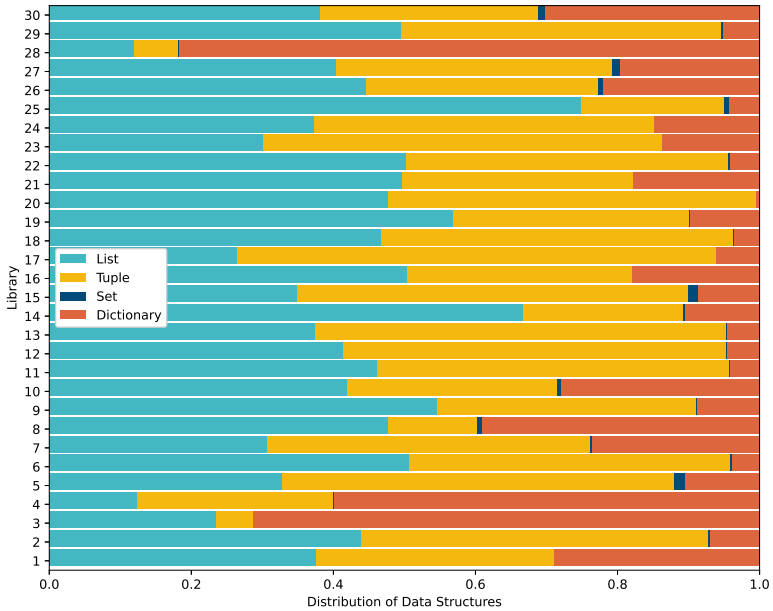


Figure 22: The probability distribution of data structures for each library. *Sets* are consistently less used than *lists*, *tuples* and *dictionaries*.

4.9 Files and Exceptions

This section, presents the results of our analysis concerning the location and correlation of keywords related to files and exceptions. Specifically, the Python keywords in focus are *raise*, *try*, *except*, *finally*, and *with*.

4.9.1 *raise*

Figure 23 shows the probability distribution of the keyword *raise* in code blocks, across all libraries. We observe that the majority of the occurrences of the *raise* keyword is found within *if* statements (0.55), followed by *global* scope (0.30) and *functions* (0.16). It is noteworthy that the occurrence of *raise* within *for* loops is very low at 0.00033, and that there are no instances of *raise* within *while* loops.

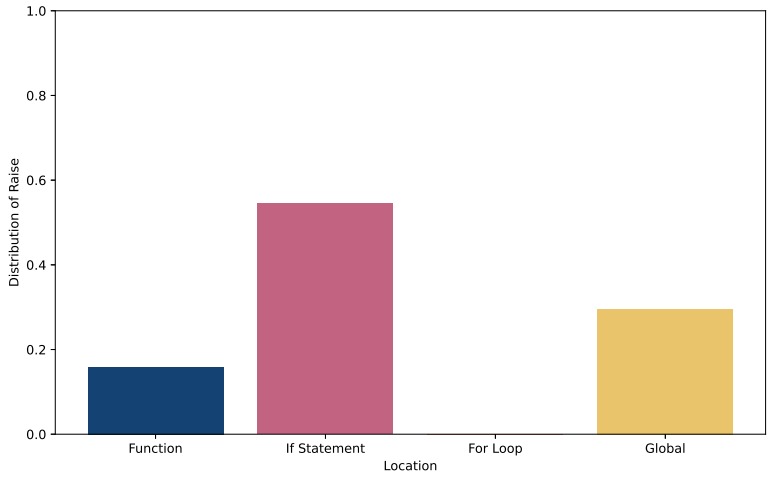


Figure 23: The probability distribution of the *raise* keyword by location for all the libraries combined. The majority of the *raise* keywords are found within *if* statements.

4.9.2 *try*

Figure 24 illustrates the probability distribution of the keyword *try* across various code blocks. Within the analyzed dataset, the majority of the *try* keyword are found within *functions* (0.54), followed by the *global* scope (0.24), *if* statements (0.12) and *for* loops (0.08), respectively. The occurrence of *try* keywords within *while* loops is minimal, with a distribution 0.02 of the total observations.

When examining the distribution per library, it becomes apparent that most libraries exhibit similar distributions. However, some libraries stand out, notably, Boto3 (library 3) and Gooey (library 8) have a particularly high proportion of *try* keywords in *functions*. Conversely, FastAPI (library 28) has a particularly low proportion of *try* in *functions*, and most of *try* keywords occur in the *global* scope, with a distribution of 0.79.

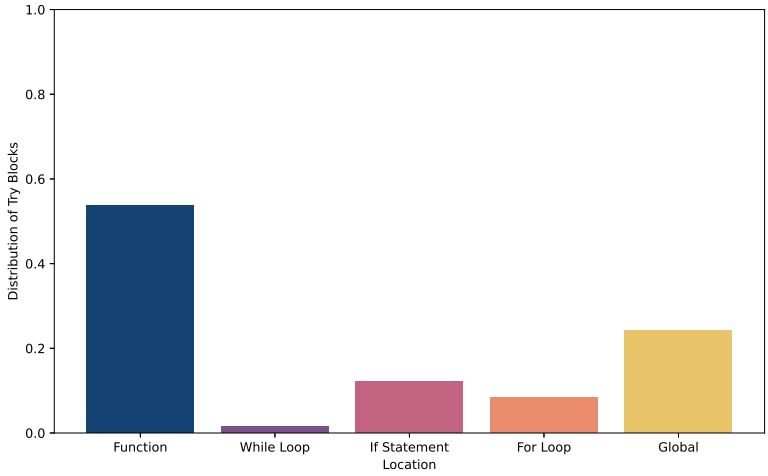


Figure 24: The probability distribution of the *try* keyword by location for all the libraries combined. The majority of the *try* keywords are found within *functions*.

4.9.3 *with*

Figure 25 displays the probability distribution of the keyword *with* in different code blocks. The distribution reveals that majority of the occurrences of the *with* keyword are located within *functions* (0.75). The remaining occurrences are divided among the *global* scope (0.15), *for* loops (0.05), and *if* statements (0.05). In no instance does the *with* keyword appear in *while* loops.

When examining the distribution per library, it is evident that most libraries have a large proportion of the *with* keywords within *functions*. Only five libraries, Gooey (library 8), NLTK (library 12), Scapy (library 21), sqlmap (library 23), and thumbor (library 24), have a proportion of the occurrence of the *with* keyword below 0.60 in functions.

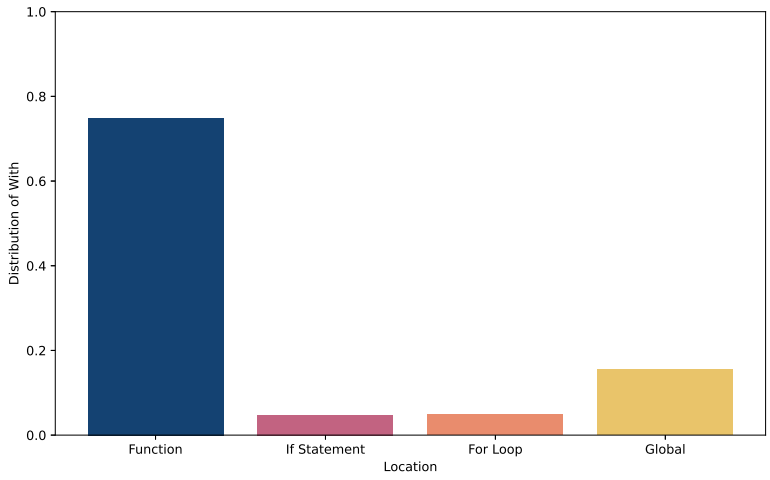


Figure 25: The probability distribution of the *with* keyword by location for all the libraries combined. The majority of the *with* keywords are found within *functions*.

4.9.4 Correlations

The correlation between the keywords *raise*, *try*, *except*, *finally*, and *with* per file is shown in Figure 26. The scatter plots depict the relationships between pairs of keywords, with the x- and y-axes representing the occurrence count of each keyword in the associated file.

The largest correlation is observed between *try* and *except*, which is expected since these keywords are commonly used together. The second largest correlation exists between *raise* and *except*, as well as between *try* and *raise*. This correlation is logical considering that *raise* is often used to raise an exception.

The correlation between *try* and *finally*, and between *except* and *finally*, is also significant but lower compared to the *try* and *except* correlation. The keyword *finally* is typically used in a *try-except* clause to perform specific operations when the conditions specified by *try* and *except* fail.

There is an insignificant correlation observed between *with* and *raise*, as well as between *with* and *except*. Finally, Table 19 presents the numerical values of the observed Pearson's correlation coefficients.

Table 19: The Pearson correlation coefficients indicates the correlation between the *try*, *finally*, *except*, *with* and *raise*.

	raise	**try**	**except**	**finally**	**with**
raise	1.000	0.469	0.474	0.128	0.002
try	0.469	1.000	0.965	0.368	0.028
except	0.474	0.965	1.000	0.187	-0.004
finally	0.128	0.368	0.187	1.000	0.146
with	0.002	0.028	-0.004	0.146	1.000

Figure 26: The correlation plots for each keyword pair *raise*, *try*, *except*, *finally* and *with*. There is one data point per file, representing the correlation between two of the keywords in that specific file. There is a strong, positive correlation between *try* and *except*.

4.10 Classes

This section, introduces the findings of our analysis regarding the average number of lines in *classes* per file, as well as the location of *classes* within the code.

Figure 27 depicts the distribution of average *class* length per file. The distribution demonstrates a positive skew, with the majority of the data concentrated towards the shorter class lengths. Table 20 presents the mean, standard deviation, median, minimum, and maximum number of lines in a class. The mean line count is 124, with a standard deviation of 156, and the median is 64 lines per class. Notably, the line count ranges from 2 to 1587, with 87% of the *classes* consisting of 280 lines or fewer. For the purpose of visualization, Figure 27 excludes the very few long class lengths.

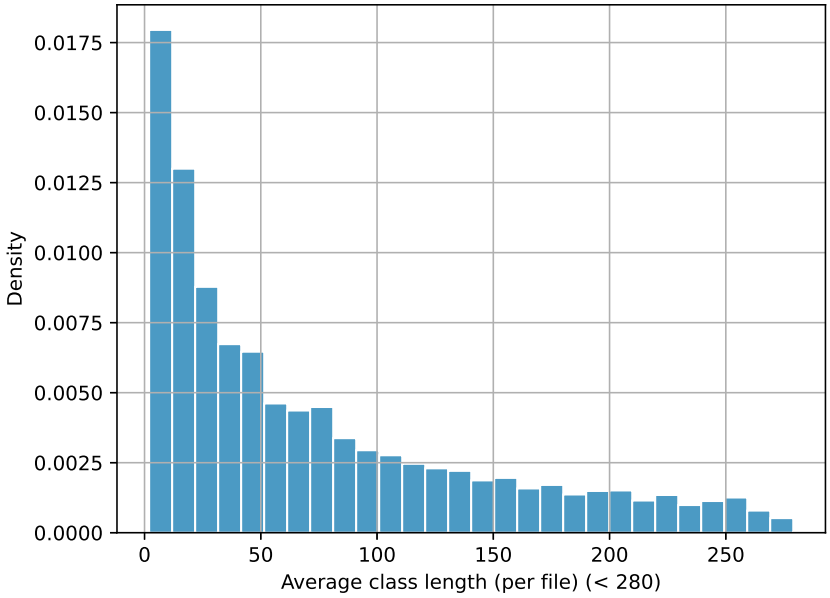


Figure 27: Average number of lines in *classes* per file. We note that for the purpose of visualization, the data outliers have been truncated.

Table 20: Statistical summary of the average number of lines in *classes* per file.

Mean	**Std**	**Median**	**Min**	**Max**
124	156	64	2	1587

Figure 28 presents the distribution of *class* locations. We observe that the majority of *classes* are defined within the *global* scope, followed by *classes* defined within *functions*, *classes*, *if* statements, and *for* loops. It is important to note that no *classes* were found within *while* loops. The analysis of cases where *classes* are defined within *functions*, *if* statements, and *for* loops is considered in the future work section. Finally, to further evaluate the distribution of *class* locations, we analyzed the per library distributions, shown in Figure 29.

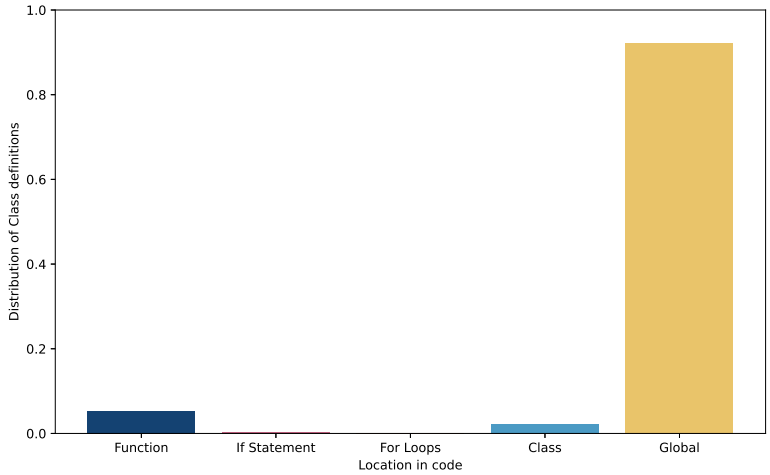


Figure 28: The probability distribution of *class* location, across all libraries. The exact probabilities are: *global* 0.92, *function* 0.05, *class* 0,02, *if statement* 0.003, *for loop* 0.0001

Overall, *classes* are primarily defined within the *global* scope. Figure 29 reveals that 20 of the 30 libraries incorporate *classes* within *functions*, 15 libraries contain nested *classes*, and 6 libraries have *classes* within *if* statements. Among the analyzed libraries, Cerberus (library 4) demonstrates a nearly equal distribution between *classes* defined in *functions* and the *global* scope, while five libraries confine all their *classes* to the *global* scope. It is worth noting that only two libraries, Numba (library 13) and Django (library 27), exhibit instances of *classes* within *for* loops, totaling four occurrences.

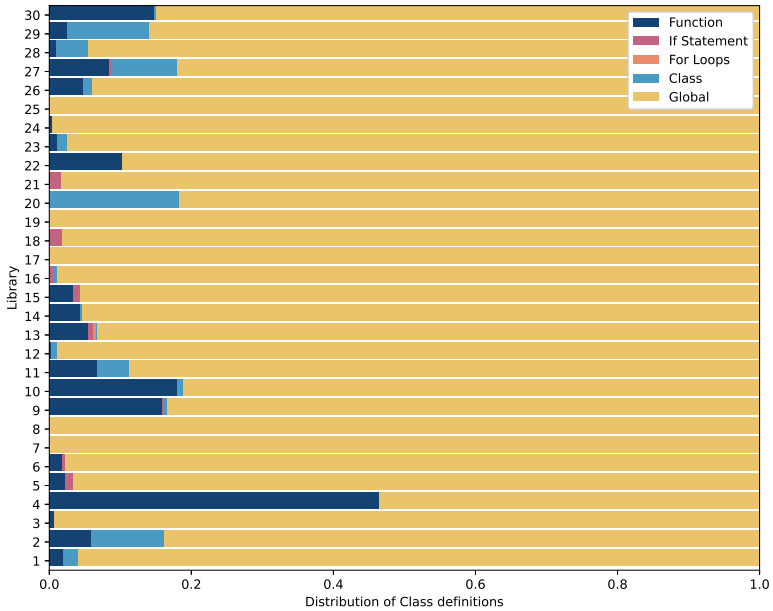


Figure 29: The probability distribution of location of *class* definitions across each library. The majority of *classes* are defined within the *global* scope.

4.11 Administrative and Engineering Results

The collaboration within our team worked well, as the members had previously worked together on projects and were familiar with each other's working methods. To maintain effective communication and progress, we planned frequent meetings. For project management, we utilized Toggle Track for time tracking and Jira for task management. Initially, Jira provided structure and clarity, but as our understanding of the project grew, regular status updates became sufficient. Further, we allocated specific responsibilities, allowing the group to work both collaboratively and independently. The distribution of responsibilities was as follows: Amalie took the lead on AST, graphics, and visualizations, while Kristina had the main responsibility of structuring the feature variables, and of PCA, and clustering analysis. Despite the allocation of specific responsibilities, both team members worked collaboratively on all aspects.

Our collaboration with the supervisor throughout the project was constructive and valuable. Initially, we held weekly meetings, but as the project progressed, we gradually increased the frequency of our meetings. We received valuable feedback, while also providing constructive challenges when we disagreed.

The project was divided into three phases: research and planning, analysis and visualization, and report. In the initial phase, we familiarized ourselves with the project, conducted relevant research, and planned accordingly. The second phase involved organizing the data, and the practical implementation, utilizing Python to generate diverse charts from the structured data. Additionally, we analysed the actual dimensionality of the dataset. During the final phase, we analyzed the results and completed the majority of the report. Each phase was completed within the planned timeframe.

Prior to the project commencement, we established result and impact targets, which are outlined in the Pre-Project Plan (Attachment A). The result target, "identify and visualize qualities of professional Python code," was successfully achieved. As for the impact targets, our aim was to enhance programming instruction. While the project's ultimate influence is yet to be determined, it is reasonable to expect that the attained results will aid novice programmers in their journey towards expertise, with the potential for utilization in NTNU courses.

For further information on planning, goals and administration, please refer to Attachment A and B.

5 Discussion and Conclusion

This project was driven by three research questions, and in this section, we discuss the findings for each question.

Firstly, we investigated whether the dataset, consisting of 30 distinct libraries and 11,712 files, could be considered a cohesive and unified set. To assess this, we conducted a dimensional analysis and attempted to cluster the files using the k-means algorithm. The results revealed that 181 dimensions were required to accurately represent the data with a 99.9% level of accuracy. Interestingly, these dimensions aligned with the number of nodes found in the AST. Moreover, we observed that the dataset could not be effectively clustered into more than two groups. Based on these findings, we can conclude that the dataset is indeed coherent and suitable for further analysis.

Our second research question focused on identifying the characteristics of professional Python code through data analysis. Several conclusions were drawn from this analysis. Firstly, functions emerged as the primary building blocks of source code. Additionally, we noted that code was predominantly organized into concise blocks comprising *functions*, conditions, and loops. Furthermore, the usage of *lists*, *tuples*, and *dictionaries* was more prevalent compared to *sets*, and *for* loops were more commonly employed than *while* loops. Finally, the logical operator == stood out as the most frequently used operator in the dataset.

Lastly, we explored whether the qualities of professional Python code can be used to guide the education of undergraduate programmers. The results section highlights key priorities that can shape programming curricula. Emphasizing the mastery of *functions* and the effective division of code into manageable blocks is crucial for novice programmers. Additionally, aspects such as object-oriented programming (OOP), control flow, error handling, and the usage of external libraries should be prioritized. These insight can provide valuable guidance for further development for programming courses.

6 Future work

Our examination of the dataset has uncovered several intriguing aspects that could be further explored. These aspects include:

- Further analysis of the two clusters identified in Section 4.2 could shed light on the type of files in each cluster.
- Analysis of libraries that deviate from the overall trends. For instance, our analysis has revealed that libraries such as Cerberus (library 4), algorithms (library 25), and FastAPI (library 28) often stand out. Exploring the factors that contribute to these deviations could uncover underlying connections or patterns associated with these libraries.
- Deeper analysis of *function* definitions, particularly the rare instances of functions inside *for* loops, *if* statements, and nested *functions*. Similarly, further investigation of the limited occurrences of *classes* nested within *classes*, *functions*, *if* statements, and *for* loops.
- Further analysis of the correlation between keywords.
- Examining the content of short *classes*, and subdividing *classes* into superclasses, abstract classes, etc.

This list does not encompass all the potential avenues for future work. As the project advances, we anticipate that additional opportunities for further development will arise.

7 Societal Impact and Sustainability

When considering the project's sustainability and societal impact, we could draw a connection to ambitious political goals, such as the United Nations' ninth sustainable development goal on "Quality education" (United Nations, n.d.). However, it is important to acknowledge that this project is a minimal contribution towards this goal, with no direct impact. Therefore, we choose to focus on the actual impact of our results, and the social group to which this project is addressed; students and teachers of programming.

The project has provided valuable insights into the characteristics of professional Python code, that are concrete and easily understandable. In a conversation with computer science professor at NTNU, Majid Rouhani, the possibilities of using the results in the design of programming courses were discussed. For example, based on the results regarding keywords, one can determine which ones are most commonly used and should be taught first. Functions can play a more central role in general, but particularly early in the course, as they are a fundamental building block of professional Python code. Through further discussions with lecturers, it became evident that some of the findings were unexpected and surprising to them. Consequently, these findings have the potential to influence the curriculum and improve the quality of programming courses.

There are several ethical considerations to explore regarding the project. For instance, is it ethically acceptable to gather knowledge from source code without obtaining permission? In our case, we utilized open-source source code, which allows modifications and redistribution (*The Open Source Definition* n.d.). Another question to ponder is whether the data is sufficiently extensive. Is it ethically sound to draw conclusions based on this dataset? Despite the diverse range of purposes represented by the libraries, the clustering results revealed a cohesive grouping among them.

Reflecting on our conduct throughout this project, we maintained a commitment to academic honesty and respect for intellectual rights by diligently citing all sources used. Further, the collaboration in the group was characterized by an open and respect full dialogue, shared responsibilities, integrity and professionalism.

In order to enhance the project's impact and inform decision-making, a valuable approach would have been to proactively engage first-year students enrolled in programming courses. This could have been achieved through surveys or interviews, providing valuable insights and lending greater credibility to the project. While the project excels in disseminating knowledge, it's important to acknowledge that the library source code may not fully represent Python code's overall style. Furthermore, it is worth noting that there are additional aspects within the project that could have been further investigated. It is our aspiration that further research will yield a comprehensive framework and structure for professional coding practices. A discussion of these possibilities is provided in Section 6.

References

Abdi, H. and Williams, L. J. (2010). "Principal component analysis". In: *Wiley interdisciplinary reviews. Computational statistics* 2.4, pp. 433–459.

Baxter, I. D. (2016). "Convolutional neutral networks over tree structures for programming language processing". In: *AAAI* 2, p. 4.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). "Latent dirichlet allocation". In: *Journal of machine Learning research* 3, pp. 993–1022.

Chi, M. T. H., Feltovich, P. J., and Glaser, R. (1981). "Categorization and Representation of Physics Problems by Experts and Novices". In: *Cognitive science* 5.2, pp. 121–152.

cpprefeence.com (n.d.). *C++ keywords*. URL: `https://en.cppreference.com/w/cpp/keyword`. (accessed: 09.06.2023).

Deerwester, S. et al. (1990). "Indexing by latent semantic analysis". In: *Journal of the American society for information science* 41, p. 391.

Gaddis, T. (2019). *Starting out with Python*. New York: Pearson. ISBN: 978-1-292-22575-3.

Han, J., Pei, J., and Kamber, M. (2011). *Data Mining: Concepts and Techniques*. Saint Louis: Elsevier Science (The Morgan Kaufmann Series in Data Management Systems). ISBN: 9781558609013.

Horstmann, C. S. and Necaise, R. D. (2019). *Python for Everyone*. Hoboken, New Jersey: Wiley. ISBN: 9781119638292.

Januzaj, Y., Beqiri, E., and Luma, A. (2023). "Determining the Optimal Number of Clusters using Silhouette Score as a Data Mining Technique". In: *International Journal of Online and Biomedical Engineering* 19.4, pp. 174–184.

Jolliffe, I. T. (1986). *Principal Component Analysis*. New York, NY: Springer Series in Statistics.

Kluyver, T. (n.d.). *Meet the Nodes*. URL: `https://greentreesnakes.readthedocs.io/en/latest/nodes.html`. (accessed: 03.04.2023).

Lee, S. et al. (2016). "Comparing Programming Language Comprehension between Novice and Expert Programmers using EEG Analysis". In: *2016 IEEE 16th International Conference on Bioinformatics and Bioengineering (BIBE)*, pp. 350–355.

Linn, M. and Clancy, M. (1992). "The case for case studies of programming problems". In: *Communications of the ACM* 35, pp. 121–132.

Mayrhauser, A. von and Vans, A. M. (1994). "Program Understanding - A Survey". In.

Mou, L. et al. (2016). "Convolutional neutral networks over tree structures for programming language processing". In: *AAAI* 2, p. 4.

Oracle Corporation (2021). *The Java® Language Specification, Java SE 17 Edition*. URL: `https://docs.oracle.com/javase/specs/jls/se17/html/jls-3.html#jls-3.9`. (accessed: 09.06.2023).

Ortin, F. et al. (2020). "Heterogeneous tree structure classification to label Java programmers according to their expertise level". In: *Future generation computer systems* 105, pp. 380–394.

Panichella, A. et al. (2013). "How to effectively use topic models for softeare engineering tasks? an approach baed on generic algorithms". In: *International Conference on Software Engineering (ICSE)* 35, pp. 522–531.

Python Software Foundation (2021a). *AST - Abstract Syntax Trees*. URL: `https://docs.python.org/3/library/ast.html`. (accessed: 03.04.2023).

— (2021b). *Python 3 Documentation - Lexical Analysis: Keywords*. URL: `https://docs.python.org/3/reference/lexical_analysis.html#keywords`. (accessed: 09.06.2023).

Språkrådet (n.d.). *Antallet ord i norsk*. URL: `sprakradet.no/svardatabase/sporsmal-og-svar/antall-ord-i-norsk/`. (accessed: 07.06.2023).

Strang, G. (2009). *Introduction to Linear Algebra*. 4th. Wellesley-Cambridge Press. ISBN: 978-09-80-23271-4.

T. Kamiya, T., Kusumoto, S., and Inoue, K. (2002). "CCFinder: a multilingustic token-based code clone detection system for large scale source code". In: *IEEE Transavtions on Software Engerneering* 28, pp. 654–670.

The Open Source Definition (n.d.). Open Source Initiative. URL: `https://opensource.org/osd/`. (accessed: 08.06.2023).

United Nations (n.d.). *Goal 4: Quality Education*. Sustainable Development Goals Knowledge Platform. URL: `https://sdgs.un.org/goals/goal4`. (accessed: 06.06.2023).

Weiser, M. and Shertz, J. (1983). "Programming problem representation in novice and expert programmers". In: *International Journal of Man-Machine Studies* 19, pp. 391–398.

Windslow, I. (1996). "Programming Pedagogy – A Psychological Overview". In: *ACM SIGCSE Bulletin* 28.3, pp. 17–25.

Zhang, J. et al. (2019). "A Novel Neural Source Code Representation Based on Abstract Syntax Tree". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794.

8 Attachments

This section contains the following attachments for the project:

- **Attachment A - Pre-Project Plan**
- **Attachment B - Project Handbook**

NTNU

Norwegian University of
Science and Technology