Doctoral Dissertation

Understanding How Developers Present Code Snippets in README Files

Supavas Sitthithanasakul
Program of Information Science and Engineering
Graduate School of Science and Technology
Nara Institute of Science and Technology

Supervisor: Kenichi Matsumoto
Software Engineering Lab. (Division of Information Science)

Submitted on August 30, 2023

A Doctoral Dissertation
submitted to Graduate School of Science and Technology,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of Engineering

Supavas Sitthithanasakul

Thesis Committee:

Supervisor	Kenichi Matsumoto (Professor, Division of Information Science)
	Hajimu Iida (Professor, Division of Information Science)
	Takashi Ishio (Professor, Future University Hakodate)
	Raula Gaikovina Kula (Associate Professor, Division of Information Science)
	Dong Wang (Assistant Professor, Kyushu University)

Understanding How Developers Present Code Snippets in README Files*

Supavas Sitthithanasakul

Abstract

As the most popular Open Source version control platform that hosts more than 330 million repositories. GitHub serves as a social coding platform that hosts services for software projects. Having newcomers continually join the projects and participate in the development process is critical for the success of the projects. One of the ways to encourage participation is through documentation.

Good software documentation is an invaluable asset to any software repository, as it helps stakeholders to use and understand software. The software repositories often release README files as a meta-file document to highlight vital information about their software. A README file plays an essential role as the initial point of contact for developers in Open Source Software (OSS) projects.

To understand content in README files, code snippets are straightforward communication to understand the functionality of the software, and may lower natural language barriers for international users. To understand the code snippets, the expertise of developers is key to comprehending idioms (i.e., code elements) found in the code snippets. Code snippets may contain both basic and proficient elements, which might pose a challenge for novice clients. While basic code snippets are generally easy to comprehend, proficient elements may be more difficult to understand for those lacking expertise. Measuring the readability of code snippets is subjective since there are various practices and styles of coding, especially in different domains of software.

This thesis presumes that the software domain is the crucial determinant in the differentiation of required competency levels for comprehending code snippets

*Doctoral Dissertation, Graduate School of Science and Technology, Nara Institute of Science and Technology, August 30, 2023.

by clients. Firstly, this thesis reveals the prevalence of the competency levels of code snippets in README files by using `pyceft`, a tool that detects and calculates the required competency level to comprehend each Python element. The results affirm prior studies by indicating that developers predominantly present basic and independent code snippets in the README files. Secondly, a quantitative analysis is conducted to investigate how developers present code snippets in README files from various software domains.

In summary, the results of this thesis highlight how developers present different competency levels of code snippets in relation to the domains of software. The key implications of this thesis comprise (i) the methodology to analyze the correlation between the competency level of code snippets and the software domain, and (ii) guidelines for developers to present code snippets that align with the appropriate competency level for their software domain.

Keywords:

Software documentation, README file, Code Competency level, Code snippet, PyPI library, Software domain

Acknowledgements

First of all, I would like to express my best sincere gratitude to my thesis supervisor, Professor Kenichi Matsumoto, for giving me opportunities to study in his laboratory as a PhD student, and also providing guidance and encouragement during my time at NAIST.

I would like to express my gratitude to Associate Professor Raula Gaikovina Kula for always providing a suggestion when I struggle in experiments, opening new areas of research, and helping me to start researching. Thank you for his patience to support me from the first day at NAIST.

To my thesis committee, I would like to thank Professor Hajimu Iida, Professor Takashi Ishio, and Assistant Professor Dong Wang, for their invaluable comments and suggestions to improve the quality of my thesis.

My special thanks go to Rungroj Maipradit, Bodin Chinthanet, and Pattaraporn Tulathum who have been like brothers and sister to me, for sharing their experiences and advice, and also having a great time together at NAIST.

I would like to thank my friends at NAIST and my labmates in the Software Engineering laboratory for a very joyful time during my PhD studying.

Last but not least, I would like to thank my family and friends in Thailand for their love, support, and encouragement to pursue my dream to study for a doctoral's degree in Japan. Without them, this thesis might not be written. Finally, I would like to express my most sincere appreciation to MEXT and NAIST staff, especially the international affair division for all kinds of support.

List of Publications

- **Do Developers Present Proficient Code Snippets in Their README Files? An Analysis of PyPI Libraries in GitHub**
  Supavas Sitthithanasakul, Bodin Chinthanet, Raula Gaikovina Kula, Natthakit Netsiwawichian, Pattara Leelaprute, Bundit Manaskasemsak, Arnon Rungsawang, Kenichi Matsumoto. Journal of Information Processing (JIP), 2023, Volume 31, Pages 679-688. (Accepted as a journal paper)

List of International Presentations

- **Do Developers Present Proficient Code Snippets in Their README Files? An Analysis of PyPI Libraries in GitHub**
  Supavas Sitthithanasakul, Bodin Chinthanet, Raula Gaikovina Kula, Natthakit Netsiwawichian, Pattara Leelaprute, Bundit Manaskasemsak, Arnon Rungsawang, Kenichi Matsumoto. 4th Workshop of the Next-Generation Software Ecosystems, Nara, Japan, Dec. 2022. (Non peer-reviewed)

- **An Analysis of Non-textual Contents in README Files in GitHub Projects**
  Supavas Sitthithanasakul, Dong Wang, Ifraz Rehman, Raula Gaikovina Kula, Kenichi Matsumoto. 2nd Workshop of the Next-Generation Software Ecosystems, Kobe, Japan, Nov. 2021. (Non peer-reviewed)

Contents

List of Figures

List of Tables

1 Introduction

As the most popular Open Source version control platform that hosts more than 330 million repositories[1]. GitHub serves as a social coding platform that hosts services for software projects, such as pull requests, and issue tracking [62, 83]. One of the advantages of a social coding platform is to gather contributions to software projects [20]. Having newcomers continually join the projects and participate in the development process is critical for the success of the projects [79]. One of the ways to encourage participation is through documentation [1].

Good software documentation is an invaluable asset to any software repository, as it helps stakeholders to use, understand, maintain, and evolve a system [3]. To advertise, attract, and possibly onboard interested developers to the projects, the software repositories often release README files as a meta-file document to highlight vital information about their software. GitHub suggests README files follow a specific format in order to help developers quickly locate important information [43].

A README file plays an essential role as the face of a software project and the initial point of contact for potential users and contributors in Open Source Software (OSS) projects. According to GitHub, a README file is a starting point to inform other developers why the project is useful, what they can do with the project, and how they can use it [43]. With a markdown syntax, developers can put visual content such as images, videos, or code snippets rather than only texts in the README files [42]. While using various visual contents, it is important for developers to utilize them and keep their README files clean to increase project accessibility and attractiveness [1, 62, 83].

[1] https://github.com/about

To understand content in README files, Liu et. al. [62] state that code snippets are straightforward communication to understand the functionality of the software, and may lower natural language barriers for international users. A code snippet is one of the visual contents that demonstrate how software and APIs should be used [41, 71, 86]. Providing simple code snippets is preferable to clients (i.e., developers or users that use the software) who might struggle with how to use the software [15, 16].

To understand the code snippets, Robles et al. [89] stated that the expertise of developers is key to comprehending idioms (i.e., code elements) found in the code snippets. Code snippets may contain both basic and proficient elements, which might pose a challenge for novice clients. While basic code snippets are generally easy to comprehend, proficient elements may be more difficult to understand for those lacking expertise. Prior study implies that code snippets should be readable with easy understandability, and reasonable complexity for code comprehension [7]. Other studies show that proficient code is written to improve the readability and execution performance of software [54, 107]. Measuring the readability of code snippets is subjective since there are various practices and styles of coding, especially in different domains of software [77, 103].

Based on the argument about how proficiency of code snippets in the README files should be. This became my motivation to reveal the usage of different competency levels of code snippets in README files and the domain of software. I hypothesize that **README files from each software domain comprise different ratios of competency levels of code snippets, which are correlated with the level of proficiency required for clients to comprehend the code elements**. In this thesis, I conduct qualitative analysis with the goals of investigating (i) the prevalence of different competency levels of code snippets in README files, and (ii) whether the software domain affects how developers present different usage of code snippets in README files.

1 Contributions

The main contributions of this thesis can be listed as follows.

- A qualitative study on the identifying competency level of code snippets in

README files.

- The empirical study to analyze the correlation between the usage of code snippets in README files and the topics of PyPI libraries.
- The replication package and dataset of this thesis which available at https://doi.org/10.5281/zenodo.7273175.

2 Organization of Thesis

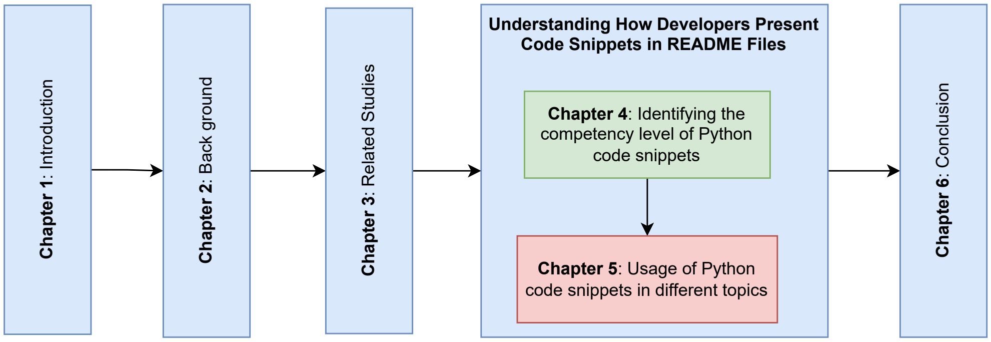

Chapter 1: Introduction
Chapter 2: Back ground
Chapter 3: Related Studies
Understanding How Developers Present Code Snippets in README Files
Chapter 4: Identifying the competency level of Python code snippets
Chapter 5: Usage of Python code snippets in different topics
Chapter 6: Conclusion

Figure 1.1. An overview of the scope of the thesis.

In this section, I describe an outline of this thesis. Figure 1.1 illustrates the structure of the thesis. The details of the rest of this thesis are listed as follows:

- **Chapter 2** presents the background of this thesis which comprises (i) the importance of README files, (ii) the competency level of code snippets, and (iii) the case study of this thesis.
- **Chapter 3** discusses the key studies that are related to this thesis.
- **Chapter 4** presents a quantitative study to identify the competency level of code snippets in README files.
- **Chapter 5** presents an analysis of the correlation between the competency level of code snippets in README files and the topics of PyPI libraries.
- **Chapter 6** concludes the study in this thesis, and direction for future works.

2 | Background

The purpose of this chapter is to describe the background and case study of this thesis. Section 1 introduces GitHub as a social coding platform. Section 2 introduces the importance of software documentation. Section 3 introduces the README file and the contents inside. Section 4 describes the importance of code snippets in the README file. Finally, Section 5 describes the PyPI library which is a case study of this thesis and the tool for calculating competency levels required to comprehend Python code snippets.

1 Social Coding Platform

As a social coding platform, GitHub is the most popular Open Source version control platform that serves as a repository for hosting services for software projects, such as pull requests, issue tracking, and work processes [62, 83]. At the time of this study, GitHub hosted more than 330 million repositories and served over 100 million developers[1].

One of the advantages of a social coding platform is to gather contributions to software projects from members of the community [20]. Having newcomers continually join the projects and participate in the development process is critical for the success of the projects [79]. To increase successful of the projects, they must find ways to encourage new developers to contribute to the development activities of the project. One of the ways to encourage participation is through documentation [1].

[1] https://github.com/about

2 Software Documentation

Good software documentation is an invaluable asset to any software repository, as it helps stakeholders to use, understand, maintain, and evolve a system [3]. There are various artifacts of software documentation such as code snippets [66, 67, 68, 105], classes and methods [34, 72, 90], code comments [95], test cases [56, 57, 78], database schemas [59], user reviews [25], user stories [49], commit messages [19, 45, 60], release Notes [73, 74], and bug reports [51, 63, 85] However, a large body of research pointed out that software documentation suffers from insufficient and inadequate content [2, 87], obsolete and ambiguous information [2, 102], and incorrect and unexplained examples [2].

To advertise, attract, and possibly onboard interested developers to Open Source Software (OSS), the software repositories often release README files as meta-file documents to highlight vital information about their software. It is often the first item that a visitor will see when visiting the repository. According to GitHub [43], it is a starting point to inform other visitors why the repository is useful, what they can do, and how they can use it.

3 README File

When accessing the repository in GitHub, the README file is one of the first things that the user encounters. It contains a free-form description of the repository and can be considered a front page for the repository [48]. The README files play an essential role in shaping a developer's first impression of a software repository and in documenting the software project that the repository hosts [83]. Informative README files have various beneficial purposes.

Figure 2.1 shows an example of README file from the software repository 'Elasticsearch Python Client' [2]. Most of the README files are created in a Markdown (MD) format. Developers (i.e., authors of the README files) might present visual content such as images, videos, or code snippets rather than only texts [42]. While using various visual contents, it is important for developers to utilize them and keep their README files clean to increase project accessibility

[2] https://github.com/elastic/elasticsearch-py

and attractiveness [1, 62, 83].

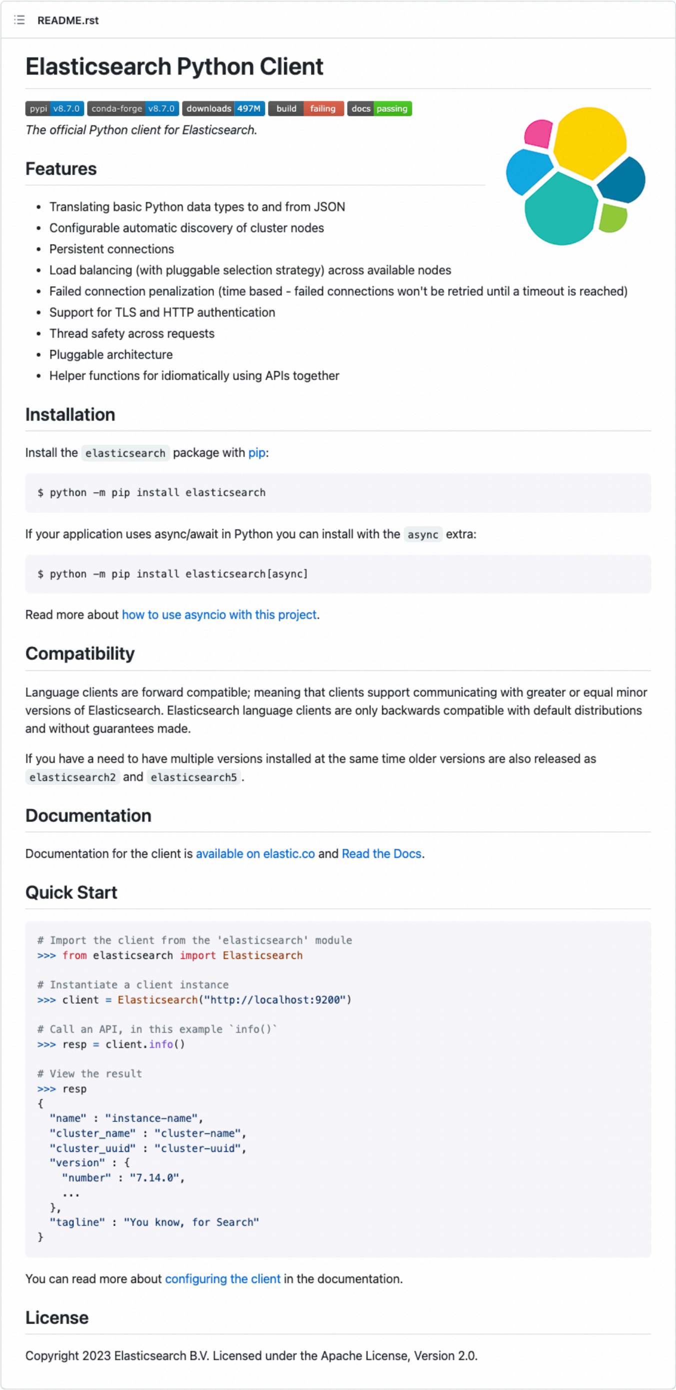
One of the primary purposes of the README file is to serve as a tutorial for getting started. The README file usually contains various information about a software repository such as overview, installation, usage, features, and license [71, 83]. After reading through the README file, clients are expected to be able to install and use the library as quickly as possible [41]. However, we sometimes found that some software repositories present empty content in their README files or do not even create it.

Related studies found that software repositories with quality README files tend to be more popular and sustainable [11, 35, 37]. One of the interview results conducted by Qiu et al. [84] noted that *"a good README allows [one] to understand what this project is about, how to install it, and how to use it. It also gives examples of code snippets for its API and their effects".* The study by Liu et al. [62] also found that software repositories that contain the usage sections in README files tend to be more popular (i.e., have a higher median number of stars). However, previous studies reported that developers do not always follow best practices or recommendations to create README files [13, 26]

4 Code Snippets in README File

In the usage sections of README files, developers usually provide code snippets to visually explain and demonstrate the usage of their software repositories [22]. For instance, in the Quick Start section in Figure 2.1, a code snippet is one of the visual contents that demonstrate how software and APIs should be used [41, 71, 86]. Before using the software, it is necessary to comprehend the code snippets that are presented in the README files. Thus, the understandability of code snippets is a crucial aspect that might significantly influence program comprehension efforts [92]. There are various ways to write code snippets, developers could use elements that do not require a deep comprehension, especially for educational purposes [65]. As shown in the prior studies [15, 16, 94], clients prefer code snippets that are easy to understand because it is suitable for novices to get started.

However, some README files might contain advanced code snippets. In par-



Elasticsearch Python Client

The official Python client for Elasticsearch.

Features

- Translating basic Python data types to and from JSON
- Configurable automatic discovery of cluster nodes
- Persistent connections
- Load balancing (with pluggable selection strategy) across available nodes
- Failed connection penalization (time based - failed connections won't be retried until a timeout is reached)
- Support for TLS and HTTP authentication
- Thread safety across requests
- Pluggable architecture
- Helper functions for idiomatically using APIs together

Installation

Install the `elasticsearch` package with pip:

```
$ python -m pip install elasticsearch
```

If your application uses async/await in Python you can install with the `async` extra:

```
$ python -m pip install elasticsearch[async]
```

Read more about how to use asyncio with this project.

Compatibility

Language clients are forward compatible; meaning that clients support communicating with greater or equal minor versions of Elasticsearch. Elasticsearch language clients are only backwards compatible with default distributions and without guarantees made.

If you have a need to have multiple versions installed at the same time older versions are also released as `elasticsearch2` and `elasticsearch5`.

Documentation

Documentation for the client is available on elastic.co and Read the Docs.

Quick Start

```
# Import the client from the 'elasticsearch' module
>>> from elasticsearch import Elasticsearch

# Instantiate a client instance
>>> client = Elasticsearch("http://localhost:9200")

# Call an API, in this example `info()`
>>> resp = client.info()

# View the result
>>> resp
{
  "name" : "instance-name",
  "cluster_name" : "cluster-name",
  "cluster_uuid" : "cluster-uuid",
  "version" : {
    "number" : "7.14.0",
    ...
  },
  "tagline" : "You know, for Search"
}
```

You can read more about configuring the client in the documentation.

License

Copyright 2023 Elasticsearch B.V. Licensed under the Apache License, Version 2.0.

Figure 2.1. An example of README file from the software repository 'Elasticsearch Python Client'

ticular, when developers are more concerned about the performance and security of their software [23]. In the case of Python code writing style, the Python developer community prefers to write code in the 'Pythonic' way, which is more elegant and readable code [4]. However, elements written in Pythonic code seem to require a deep knowledge of the language [89]. Especially on some libraries that are huge and comprise advanced code snippets which sometimes provide more functionalities than clients needed [52].

5 Case Study: Python Package Index (PyPI) Libraries

To validate my thesis statement, it is necessary to analyze (i) how developers present code snippets with different competency levels in README files, and (ii) whether the domains of software correlate with the usage of code snippets. However, to the best of my knowledge, there is no prior study on these points. Since there are various categorizations for differentiating each software, e.g., programming languages, architectures, purposes of software usage [46]. So, to properly and accurately validate the correlation between the competency level of code snippets in README files from different software domains, I decided to use README files of Python libraries from the Python Package Index (PyPI) as a case study in this thesis.

5.1 pycefr: A tool for calculating competency level of Python code snippets

Nowadays, Python is one of the most used programming languages [17]. It is known to be versatile and suitable for a wide range of people, from novices to experienced developers [89]. From an example of the Python code snippet in Figure 2.1, it comprises various Python elements connected to each other. The Python elements can be various types of statements or constructs [89], e.g., data types, containers, variable assignments, branching and looping control, functions, exceptions and tracebacks, and classes [29].

The Python code snippets can be written in different ways depending on the

Table 2.1. Competency level of Python elements based on `pycefr` [89]

Competency Level	**Example of Python elements**
A Basic level	Print, If statement, List, Open function (files), Nested list
B Independent level	List with a dictionary, Nested dictionary, with, List comprehension, __dict__attribute
C Proficient level	__slots__, Generator function, Meta-class, Decorator class

programming experience of developers. In the README files, clients might encounter code snippets that are composed solely of basic elements or that combine both basic and advanced elements. To assist clients in estimating the proficiency level required to comprehend Python code snippets, Robles et al. [89] created `pycefr`, a tool that analyzes a given Python code snippet, then, reports the detected Python elements and calculates the competency level of each element. It was created based on the "Common European Framework of Reference for Languages: Learning, Teaching, Assessment", or CEFR [76], which is a guide to describe the performance of foreign natural language learners. As shown in Table 2.1, `pycefr` calculate Python elements into three groups of competency level (i.e., 'A': Basic, 'B': Independent, and 'C': Proficient).

5.2 Domain categorization of PyPI libraries by using the classifiers

The domains of PyPI libraries can be categorized based on behavior and characteristics, which are implemented for different purposes [53]. In the PyPI search page[3], the topic is one of the classifiers used for categorizing domains of libraries based on their characteristic. Figure 2.2, shows the topics (i.e., as highlighted by the dark gray boxes) and subtopics (i.e., as highlighted by the light gray boxes). The topic helps clients to know whether the library is created for their project or not. Currently, there are 24 topics existing. According to the PyPI classifiers

[3] `https://pypi.org/search`

page[4], these topics are usually defined by the library's developers in the 'setup.py' file which is located in the root directory of the libraries. Then the defined topics will be listed in the classifiers section. The example in Figure 2.3 shows how the topics are listed on the library page of library 'sshconf'[5] as highlighted by the gray box.

As different topics of PyPI libraries intend to serve different kinds of clients, I suspect that it may affect how developers introduce the usage of their libraries in the README file. Specifically, I hypothesize that the topic of PyPI libraries is one of the factors that affect README files from different topics to have different usage of code snippets. Based on all the above reasons, I decided that PyPI is a suitable case study to validate my thesis statement.

[4] https://pypi.org/classifiers

[5] https://pypi.org/project/sshconf

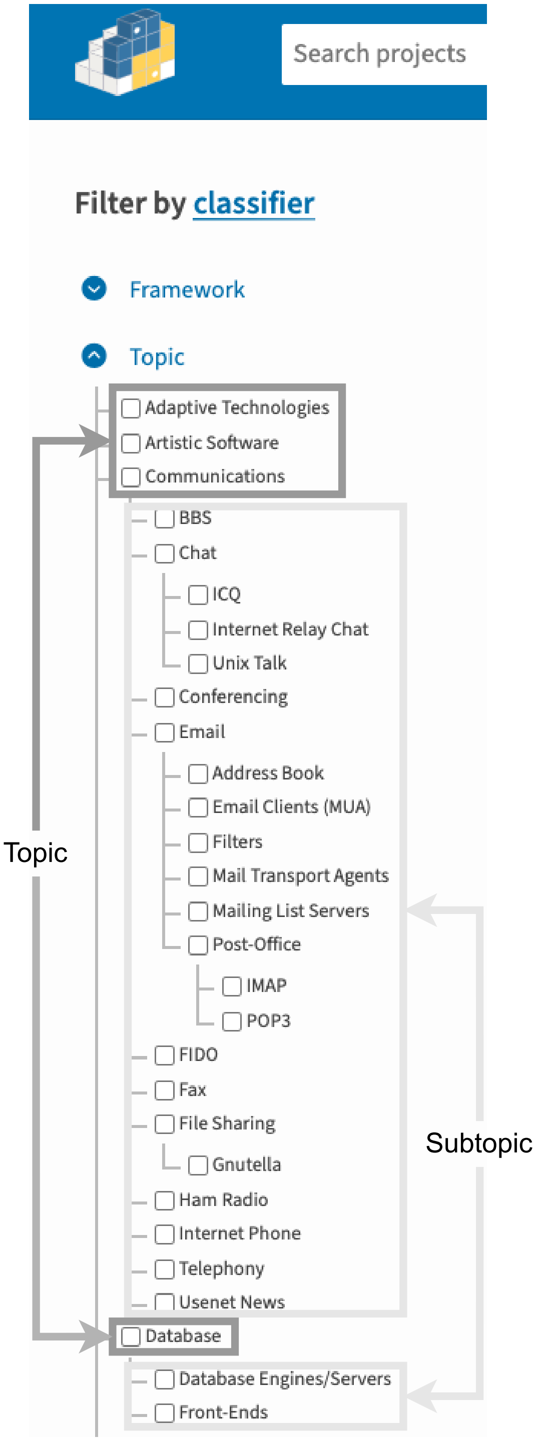
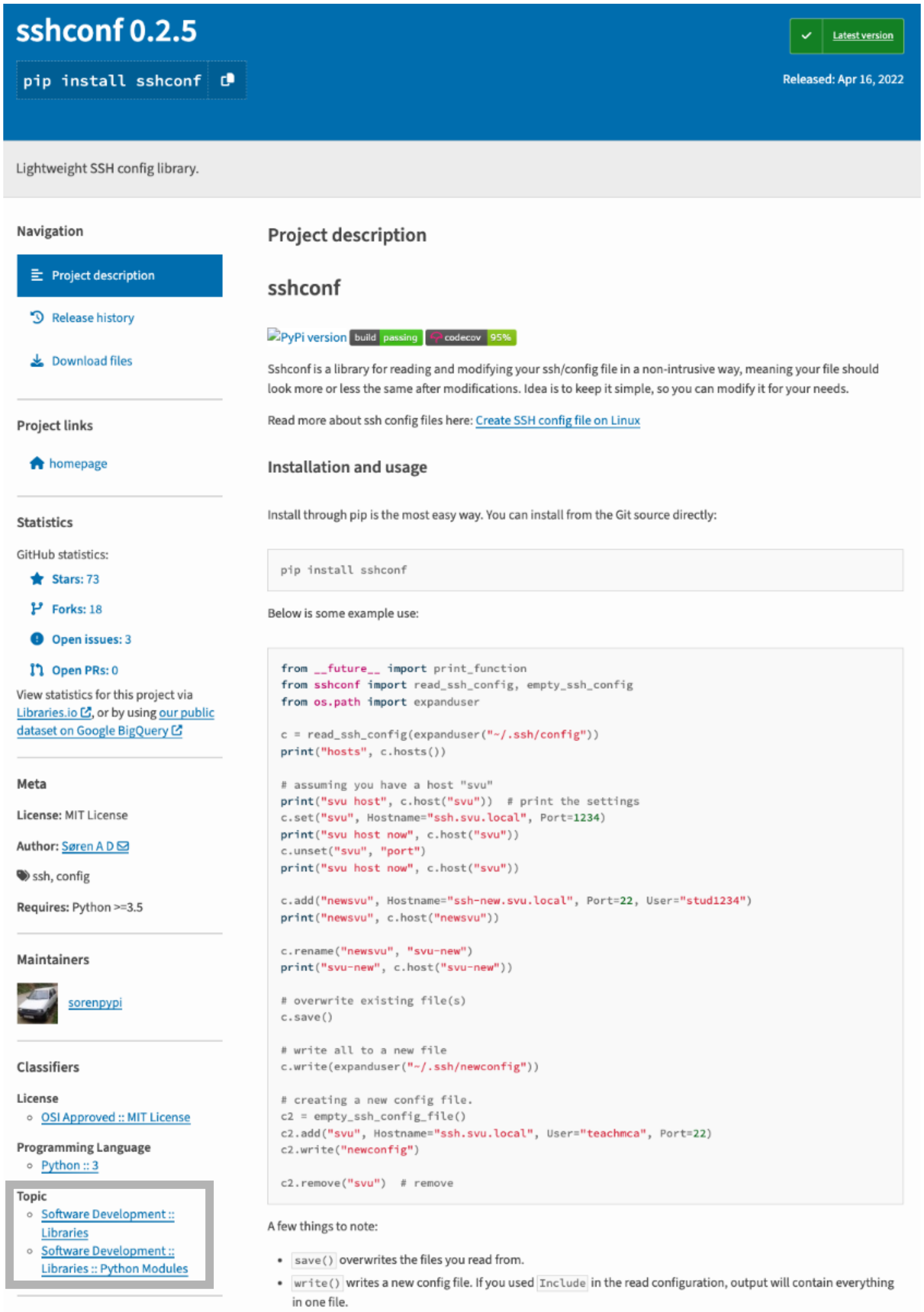


Figure 2.2. A hierarchical structure of topics of PyPI libraries



sshconf 0.2.5

```
pip install sshconf
```

Released: Apr 16, 2022

Lightweight SSH config library.

Navigation

Project links

Statistics

GitHub statistics:

- Stars: 73
- Forks: 18
- Open issues: 3
- Open PRs: 0

View statistics for this project via Libraries.io, or by using our public dataset on Google BigQuery

Meta

License: MIT License

Author: Søren A D

ssh, config

Requires: Python >=3.5

Maintainers

sorenpypi

Classifiers

License
- OSI Approved :: MIT License

Programming Language
- Python :: 3

Topic
- Software Development :: Libraries
- Software Development :: Libraries :: Python Modules

Project description

sshconf

Sshconf is a library for reading and modifying your ssh/config file in a non-intrusive way, meaning your file should look more or less the same after modifications. Idea is to keep it simple, so you can modify it for your needs.

Read more about ssh config files here: Create SSH config file on Linux

Installation and usage

Install through pip is the most easy way. You can install from the Git source directly:

```
pip install sshconf
```

Below is some example use:

```
from __future__ import print_function
from sshconf import read_ssh_config, empty_ssh_config
from os.path import expanduser

c = read_ssh_config(expanduser("~/.ssh/config"))
print("hosts", c.hosts())

# assuming you have a host "svu"
print("svu host", c.host("svu"))  # print the settings
c.set("svu", Hostname="ssh.svu.local", Port=1234)
print("svu host now", c.host("svu"))
c.unset("svu", "port")
print("svu host now", c.host("svu"))

c.add("newsvu", Hostname="ssh-new.svu.local", Port=22, User="stud1234")
print("newsvu", c.host("newsvu"))

c.rename("newsvu", "svu-new")
print("svu-new", c.host("svu-new"))

# overwrite existing file(s)
c.save()

# write all to a new file
c.write(expanduser("~/.ssh/newconfig"))

# creating a new config file.
c2 = empty_ssh_config_file()
c2.add("svu", Hostname="ssh.svu.local", User="teachmca", Port=22)
c2.write("newconfig")

c2.remove("svu")  # remove
```

A few things to note:

- `save()` overwrites the files you read from.
- `write()` writes a new config file. If you used `Include` in the read configuration, output will contain everything in one file.

Figure 2.3. Example of PyPI library ‘sshconf’ displaying how the topics are listed

3 | Related Studies

Complementary related studies are introduced throughout the thesis. This chapter describes key related works.

1 Studies on the Usage of Software Documentation

Treude et al. [97] gathered works from the data and information quality community and then propose a framework that decomposes documentation quality into ten dimensions of structure, content, and style. Lethbridge et al. [55] explored how developers use and maintain documentation, using several data-gathering approaches. The study confirmed that developers typically do not update documentation as timely or completely as software process personnel and managers advocate. Robillard et al. [88] advocated for a new vision of on-demand developer documentation, to promote automated developer documentation generation. They discussed the challenges of on-demand developer documentation in the areas of information inference, document request, and document generation. Uddin and Robillard [99] identified ambiguity, incompleteness, and incorrectness as the three most severe problems that lead API documentation to fail, based on surveys with 323 IBM software professionals. Several recent papers have analyzed open source software documentation. Hata et al. [38] investigated the characteristics of links in source code comments, the purpose of those links, and how they evolve and decay, using GitHub as a data source. Their results show that links are prevalent in source code repositories and that licenses, software homepages, and specifications are common types of link targets. Specifically, links are rarely updated, but many link targets evolve. Wattanakriengkrai et al. [101] used a mixed-method approach to analyze the role of academic paper references contained in these repositories. They found that academic papers from top-tier SE venues are not likely to reference a repository, but when they do, they usually link to a GitHub software repository.

2 Studies on the Content in README Files

Hauff et al. [39] claimed that the README files can be used to learn more about a developer's skills and interests. It is a wealth of information that the developers should be familiar with if they own a repository. Portugal et al. [82] summarized requirements-related information by using Music Application as a domain exemplar and focusing on the README file perspective of GitHub repositories. Decan et al. [24] found that 40.9% of all R packages on GitHub have a README file that contains instructions to install the package from GitHub. They analyzed these README files by using a regular expression. Sharma et al. [93] explored the possibility of developing a cataloging system by automatically extracting functionality descriptive text segments from README files of GitHub repositories. Zhang et al. [106] built a recommendation system called RepoPal to detect similar repositories based on their motivation, i.e., repositories whose README files contain similar contents are likely to be similar to one another. Hassan and Wang [36] proposed the named entity recognition(NER) technique to automatically extract software build commands from software readme files and Wiki pages, and combine the extracted commands for software building. Kelley and Garijo [47] introduced a framework for automatically extracting scientific software metadata from README files and then structuring the extracted metadata into a Knowledge Graph of scientific software. Mao et al. [64] proposed SoMEF, a software metadata extraction framework that is designed to help highlight the most important parts of scientific software documentation. This framework processes the README files in GitHub repositories and then automatically extracts which parts of their text refer to the description, installation, invocation, or citation of a software component. Greene and Fischer [32] developed CVExplorer, a tool to extract, visualize, and explore relevant technical skills data from README files in GitHub, such as languages and used libraries.

3 Studies on the Code Snippets

The code snippet is one of the crucial content in the README files. Robillard [87] found that 78% of developers learned Application Programming Interfaces (APIs) by reading documentation (i.e., README files) and 55% of them learned code snippets in the examples section. Developers are concerned about the quality of code snippets, which is one of the signals when choosing open source software projects for making contributions [84]. Mora et al. [23] found that Non-functional properties such as performance and security are important factors for developers looking to incorporate libraries in their projects, which refers to the efficiency of its underlying code snippets. Code snippets are also studied for other purposes. Kawaguchi et al. [46] proposed MUDABlue, a web tool that automatically categorizes software domains by relying on only code snippets. Zhang et al. [106] detected similar repositories by using code snippets as one of the inputs for heuristics. Alexandru et al. [4] explored how Python developers understand the term Pythonic by means of quality code, which could hold for other programming languages and ecosystems. Escobar-Avila et al. [27] proposed a novel approach to automatically categorize software by using semantic information recovered from bytecode and an unsupervised algorithm to assign categories to software systems. Linares-Vásquez et al. [58] proposed an approach to use Application Programming Interface (API) calls from third-party libraries for automatic categorization of software applications that use these API calls. This approach enables different categorization algorithms to be applied to repositories that contain both code snippets and bytecode of applications. Bajracharya et al. [8] presented structural-semantic indexing (SSI), a technique to associate words to source code entities based on similarities of API usage. Tian et al. [96] proposed LACT, a technique to automatically categorize software systems in open-source repositories. It is used to index and analyze source code documents as mixtures of probabilistic topics. Gilda [30] studied the automatic identification of the programming language by utilizing an artificial neural network based on supervised learning and intelligent feature extraction from the code snippets. Mi et al. [70] proposed DeepCRM, a deep learning-based model for code readability classification by using convolutional neural networks (ConvNets). Cao et al. [14] proposed an approach to classifying code elements in a document into contextual code elements and salient code elements. They filtered the noise traceability links between a software document and its contextual code elements to get a higher-quality link set. Baltes and Diehl [9] conducted a large-scale empirical study to analyze the usage and attribution of non-trivial Java code snippets from Stack Overflow answers in open source GitHub projects. Zhong et al. [108] developed an API usage mining framework and its supporting tool called MAPO (Mining API usage Pattern from Open source repositories) for mining API usage patterns from code snippets automatically. Baltes et al. [10] conducted a research design and summarized results of an empirical study to analyze attributed and unattributed usages of Stack Overflow code snippets in GitHub projects. Gupta et al. [33] presented JCoffee, a tool that leverages compiler feedback to convert partial Java programs into their compilable counterparts by simulating the presence of missing surrounding code snippets. Zhou and Walker [109] proposed a lightweight and version-sensitive framework to detect deprecated API usages in example code snippets on the web so developers can be informed of such usages before they invest time and energy into studying them.

4 Studies on the Software Domain

In large software repositories such as GitHub, the categorization of software assumes a crucial role, facilitating clients to browse and search for software that aligns with their specific requirements [46]. Studies conducted by Borges et al. [12] indicated that software repositories can be mainly categorized into two distinct types, i.e., domain (e.g., application software, system software, web libraries, and software tools) and programming language. Various studies have considered the software domain as a differentiating factor or as a means to identify software with similar characteristics. Linares-Vásquez et al. [61] proposed CLANdroid, a tool for automatically detecting closely related applications in Android by relying on advanced Information Retrieval techniques and five semantic anchors: identifiers, Android APIs, intents, permissions, and sensors. The results show that

CLANdroid is useful for detecting similar Android apps across different domains. Altarawy et al. [5] presented Lascad, a language-agnostic software categorization and similar application detection tool by applying Latent Dirichlet Allocation (LDA) and hierarchical clustering to reveal which software belongs to the same domain in terms of similar functionalities. McMillan et al. [69] created an approach for automatically detecting closely related applications (CLAN) which helps users detect similar applications for a given Java application. Grechanik et al. [31] offered an approach called Exemplar (EXEcutable exaMPLes ARchive) for finding highly relevant software projects from a large archive of executable applications. Hayes-Roth et al. [40] presented DSSA, a domain-specific software architecture that is developed for a large application domain of adaptive intelligent systems (AISs). It provides a reference architecture designed to meet the functional requirements shared by applications in the AISs domain. Nafi et al. [75] proposed the CroLSim model to detect similar software applications across different programming languages. They define a semantic relationship among cross-language libraries and API methods using functional descriptions and a word-vector learning model. Wang et al. [100] proposed an SVM-based categorization framework to classify the massive number of software hierarchically based on online profiles across multiple repositories.

Based on the related studies, the code snippet is one of the most important contents in the README files. To effectively utilize software repositories, clients are encouraged to fully comprehend the code snippets in the README files. In this thesis, I believe that the competency level of code snippets in README files may be affected by different characteristics of software domains. To the best of my knowledge, there is no prior work that conducted a study on how developers present code snippets with different competency levels in the README files. I believe that this study highlights the challenges and information about the correlation between competency levels of code snippets and software domains for developers, clients, and researchers.

4 Identifying the Competency Level of Python Code Snippets

1 Introduction

In software repositories, the README file is one of the most important documents that serve as a manual to use the software [83]. The well-written README files may assist clients in understanding necessary information about the software (e.g., overview, installation, example, contribution, and license) [43]. However many README files are left unattended with no or less useful information about the repositories.

Fan et al. [28] found that README files from popular repositories tend to present more code snippets as an example usage. They found that unpopular README files might not present detailed enough code snippets. The code snippet is an important element in the README file that developers can use to visually demonstrate the usage of the repositories [22].

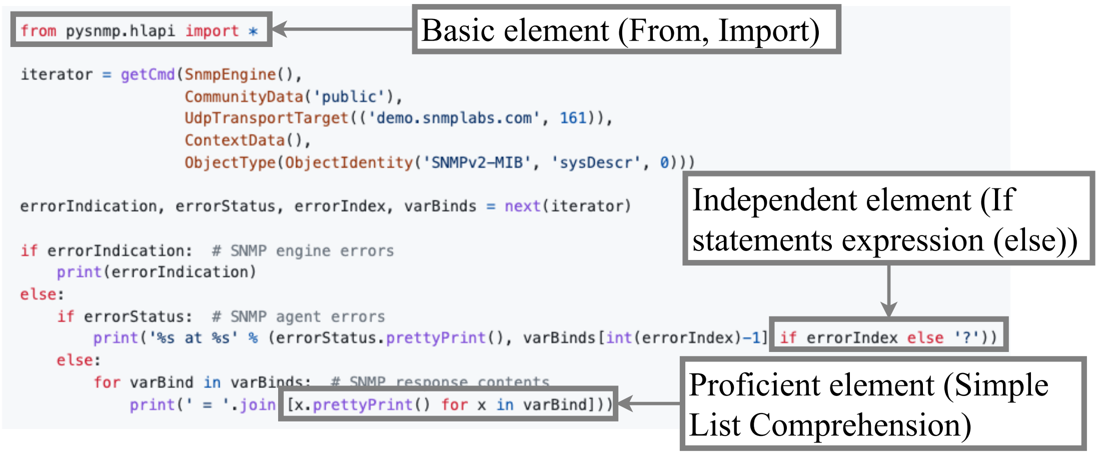
Casalnuovo et al. [15] found that clients prefer code snippets that are easy to comprehend, so they can achieve a better understanding of the repository usage. In contrast, In Python programming, Zhang et al. [107] found that developers sometimes prefer to introduce advanced code snippets (pythonic code) because these codes provide better readability and execution performance. As seen by the facts from both points of view, to the best of my knowledge, there is no related work that studies the extent of the competency level of code snippets in README files.

In this chapter, I quantitatively investigate the prevalence of the competency level of code snippets in README files. By using PyPI libraries as a case study, I extract Python code snippets from 1,620 README files of each library. As motivated by an example in Figure 4.1, Python code snippets may comprise both basic and proficient code elements. The goal of this chapter is to investigate the prevalence of different competency levels of Python code snippets in README files. By identifying the competency levels of each Python code element and then categorizing README files based on the competency level of code snippets, the results show that 45% - 50% of README files from PyPI libraries comprise independent Python code snippets while 34% - 43% comprise only basic Python code snippets. Even though the README files mainly comprise basic Python code elements. However, The popular proficient README files tend to have a balanced amount of different competency levels of code snippets while approximately half of the code snippets in the unpopular proficient README files are proficient level.

The rest of this chapter is organized as follows. Section 2 describes the methodology to identify competency levels of Python code snippets in README files. Section 3 describes the result in this chapter. Section 4 discusses the implication of this chapter. Section 5 describes threats to validity of this study. Finally, Section 6 summarizes the results of this chapter.

2 Method

To identify the competency level of Python code snippets and their prevalence in README files, I conducted a quantitative study to systematically analyze how different competency levels of code snippets are presented in README files. The study would reveal the prevalence of Python code snippets from each competency level. The overview approach study of this chapter is shown in Figure 4.2. The detail of studying is described as follows.



```
from pysnmp.hlapi import *

iterator = getCmd(SnmpEngine(),
                  CommunityData('public'),
                  UdpTransportTarget(('demo.snmplabs.com', 161)),
                  ContextData(),
                  ObjectType(ObjectIdentity('SNMPv2-MIB', 'sysDescr', 0)))

errorIndication, errorStatus, errorIndex, varBinds = next(iterator)

if errorIndication:  # SNMP engine errors
    print(errorIndication)
else:
    if errorStatus:  # SNMP agent errors
        print('%s at %s' % (errorStatus.prettyPrint(), varBinds[int(errorIndex)-1] if errorIndex else '?'))
    else:
        for varBind in varBinds:  # SNMP response contents
            print(' = '.join([x.prettyPrint() for x in varBind]))
```

Basic element (From, Import)

Independent element (If statements expression (else))

Proficient element (Simple List Comprehension)

Figure 4.1. An example of different competency levels of Python elements in Python code snippet.

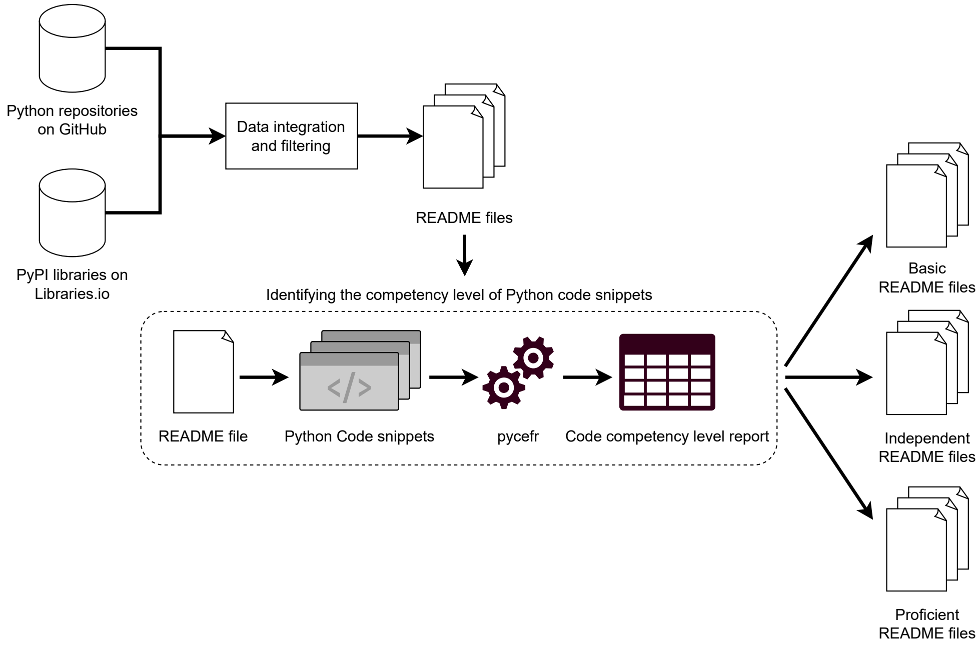

Python repositories on GitHub
PyPI libraries on Libraries.io
Data integration and filtering
README files
Identifying the competency level of Python code snippets
README file
Python Code snippets
pycefr
Code competency level report
Basic README files
Independent README files
Proficient README files

Figure 4.2. An overview approach to identify the competency level of Python code snippets

2.1 Data Preparation

In this chapter, I created the dataset by gathering README files from Github repositories that hosted PyPI libraries and then extracted Python code snippets. Table 4.1 shows a summary of the dataset in this chapter. The detail of dataset preparation is described as follows.

Step 1: Data integration and filtering. In this step, I gather a list of GitHub repositories that hosted PyPI libraries by integrating datasets from two sources. The first dataset was provided by Wattanakriengkrai et al. [101]. I contacted the authors and obtained the list of 770,217 GitHub repositories written in Python, which were created between 2014 and 2018. For the second dataset, I extract the list of PyPI libraries from the dump dataset from Libraries.io[1]. The latest updated date of the dump dataset is 2020, January 12. Next, I integrated both datasets together, which resulted in 4,612 repositories of PyPI libraries remaining. Finally, I apply the filters to exclude the following four cases: (i) the repositories do not contain README files at the root directory, (ii) the README files not written in English, (iii) the README files not contain Python code snippet, and (iv) the topic of libraries are not defined in PyPI website[2]. After filtering, there are 1,620 repositories that contain qualified README files.

Step 2: Data labelling. In GitHub, each repository has different popularity. I hypothesize that the competency level of code snippets is one of the factors that affect the popularity of the repositories. In this step, I categorize the popularity of repositories based on their number of stars. However, there is a skewness in the distribution of the number of stars, because only a minor proportion of repositories acquired a high number of stars. Following prior studies [6, 28, 81, 104], I applied the 20/80 boundary which is commonly used to split the skewed dataset. I labeled the top 20% of repositories that acquired the most number of stars as popular. For the bottom 70% of repositories, I labeled them as unpopular. The remaining 10% of repositories are labeled as a gap, so I can analyze the difference with respect to the number of stars between the two groups of repositories more considerably. From 1,620 repositories, 325 repositories that have at least 84 stars are labeled as popular, the 1,128 repositories that have at

[1] https://libraries.io/data

[2] https://pypi.org/

Table 4.1. Summary of the dataset.

Downloaded date of the README files	May, 2022
# GitHub repositories written in Python	770,217
# GitHub repositories that hosted PyPI libraries	4,612
# README files after filtering	1,620
# Python code snippets	5,511

most 38 stars are labeled as unpopular, and the rest 167 repositories are labeled as gap.

Step 3: Python code snippet extraction. In this step, first, I download the README files from each repository. Similar to the prior work [98], I then parsed the README files by using markdown format[3]. According to Fan et al. [28], in markdown format, all of the code snippets are fenced by lines with three backticks. For the Python code snippets, it can be denoted by adding the word ‘python’ after the first three backticks as shown in the example of the markdown syntax of Python code snippets in Figure 4.3. This notation is called the info_strings[4] (i.e. keyword used to specify the language of the code snippet). To extract Python code snippets, I use regular expression ˆ(\‘\‘\‘*python*$) to detect the beginning of the code snippet, then collect contents in every line until the ending of the code snippet is detected by using regular expression ˆ(\‘\‘\‘). Finally, each code snippet is then saved into a Python format file. As a result, 5,511 Python code snippets are extracted from all README files.

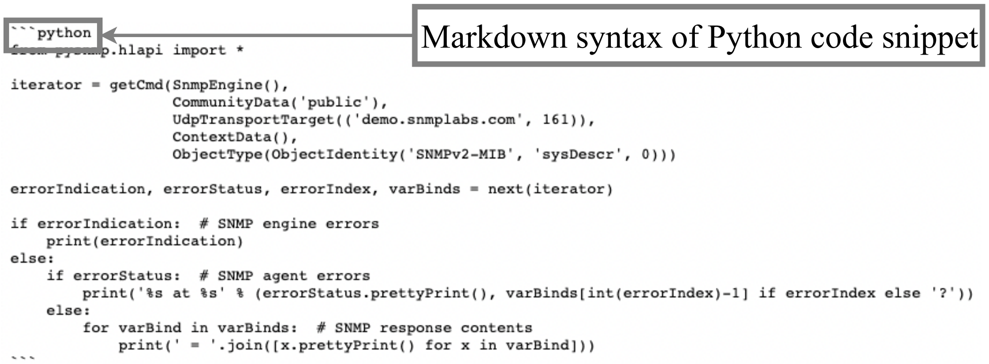
2.2 Data Analysis

In this phase, I identify the competency level of Python code snippets, then analyze the prevalence of each level. The README files are then categorized based on the highest competency level of Python elements inside the code snippets. The detail of the analysis is described as follows.

Step 1: Identification of competency level of Python code snippets.

[3] https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax

[4] https://github.github.com/gfm/#info-string



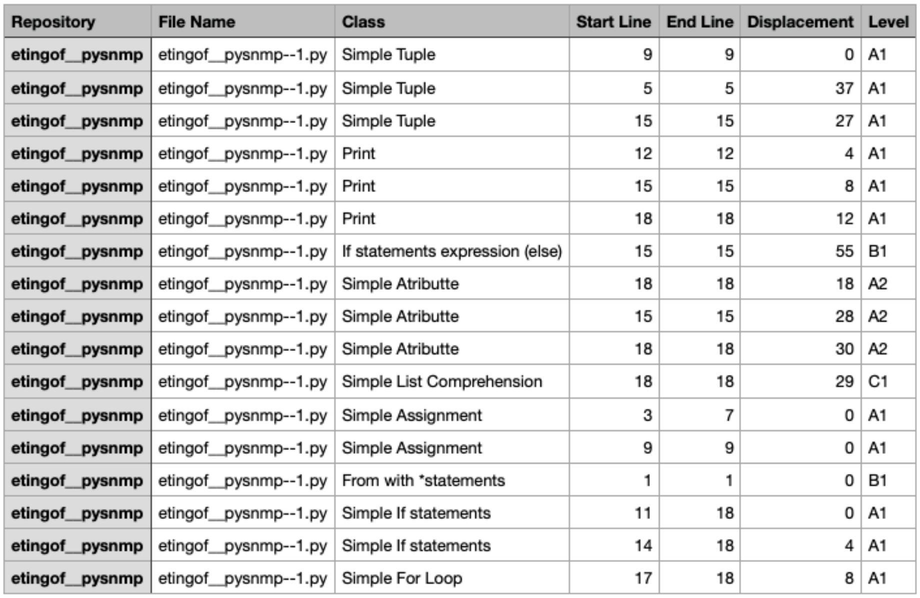
~~~
```python
from pysnmp.hlapi import *

iterator = getCmd(SnmpEngine(),
                  CommunityData('public'),
                  UdpTransportTarget(('demo.snmplabs.com', 161)),
                  ContextData(),
                  ObjectType(ObjectIdentity('SNMPv2-MIB', 'sysDescr', 0)))

errorIndication, errorStatus, errorIndex, varBinds = next(iterator)

if errorIndication:  # SNMP engine errors
    print(errorIndication)
else:
    if errorStatus:  # SNMP agent errors
        print('%s at %s' % (errorStatus.prettyPrint(), varBinds[int(errorIndex)-1] if errorIndex else '?'))
    else:
        for varBind in varBinds:  # SNMP response contents
            print(' = '.join([x.prettyPrint() for x in varBind]))
```
~~~

Figure 4.3. An example of markdown syntax of Python code snippet.

In this step, I apply `pycefr`[5] as a tool for identifying the competency level of code snippets. Each Python element in the code snippets from the Python format files will be identified and assigned a competency level. Figure 4.1 shows example of how `pycefr` detect then assign competency level to the Python elements. Figure 4.4 shows the CSV-formatted report of the competency level of Python elements that were detected in the Python code snippets. From the identification result, I assigned the competency level of each code snippet based on the highest detected competency level of Python elements. If the code snippet contains only elements with competency level 'A', I define the competency level of the code snippet as 'Basic'. If the code snippet contains at least one element with competency level 'B' or 'C', I define the competency level as 'Independent' or 'Proficient', respectively. Finally, the code snippets from the same README file are then grouped together.

Step 2: Analysis of the prevalence of different competency levels of Python elements in README files. To analyze the prevalence of each competency level, I count the number of each type of Python element from all README files and then group the elements with same competency level together. From the total number of elements from each competency level, I calculated the mean, median, standard deviation, and distribution of each competency level in the README files. In order to statistically validate the differences in the results, I applied Kruskal-Wallis non-parametric test [50] and the effect size by

[5] https://github.com/anapgh/pycefr



Repository	File Name	Class	Start Line	End Line	Displacement	Level
etingof__pysnmp	etingof__pysnmp--1.py	Simple Tuple	9	9	0	A1
etingof__pysnmp	etingof__pysnmp--1.py	Simple Tuple	5	5	37	A1
etingof__pysnmp	etingof__pysnmp--1.py	Simple Tuple	15	15	27	A1
etingof__pysnmp	etingof__pysnmp--1.py	Print	12	12	4	A1
etingof__pysnmp	etingof__pysnmp--1.py	Print	15	15	8	A1
etingof__pysnmp	etingof__pysnmp--1.py	Print	18	18	12	A1
etingof__pysnmp	etingof__pysnmp--1.py	If statements expression (else)	15	15	55	B1
etingof__pysnmp	etingof__pysnmp--1.py	Simple Atributte	18	18	18	A2
etingof__pysnmp	etingof__pysnmp--1.py	Simple Atributte	15	15	28	A2
etingof__pysnmp	etingof__pysnmp--1.py	Simple Atributte	18	18	30	A2
etingof__pysnmp	etingof__pysnmp--1.py	Simple List Comprehension	18	18	29	C1
etingof__pysnmp	etingof__pysnmp--1.py	Simple Assignment	3	7	0	A1
etingof__pysnmp	etingof__pysnmp--1.py	Simple Assignment	9	9	0	A1
etingof__pysnmp	etingof__pysnmp--1.py	From with *statements	1	1	0	B1
etingof__pysnmp	etingof__pysnmp--1.py	Simple If statements	11	18	0	A1
etingof__pysnmp	etingof__pysnmp--1.py	Simple If statements	14	18	4	A1
etingof__pysnmp	etingof__pysnmp--1.py	Simple For Loop	17	18	8	A1

Figure 4.4. An example of the CSV-formatted report of competency level of Python elements in Python code snippet by using `pycefr`.

using Cliff's δ [91]. I tested the null hypothesis that 'the amount of Python elements from difference competency levels are the same'. The interpretation of Cliff's δ is shown as follows: (i) $|\delta| < 0.147$ as Negligible, (ii) $0.147 \leq |\delta| < 0.33$ as Small, (iii) $0.33 \leq |\delta| < 0.474$ as Medium, or (iv) $0.474 \leq |\delta|$ as Large.

Step 3: Categorize README files. The objective of this step is to categorize README files based on the highest competency level of Python code snippets contained. For the README files that comprise only code snippets with competency level 'A', I define this group as 'Basic'. In the same way, if the README file comprises at least one Python code snippet with competency level 'B' or 'C', I define the group name as 'Independent' or 'Proficient', respectively.

3 Results

In this section, I analyze the results of the quantitative study of the prevalence of competency levels of Python code snippets in README files as follows.

3.1 Prevalence of Python elements in README files

From the analysis result by `pycefr`, I received a report of competency levels for each Python element in README files. To analyze the usage of Python elements in README files, I select the top five presented elements from each competency level group as shown in Table 4.2.

Overall, the basic elements (Level 'A') are the most presented elements in the README files, especially elements 'Simple Attribute' and 'Simple Assignment' (i.e., 87.22% and 86.98%, respectively). The independent elements (Level 'B') can be found at approximately 11.30% in README files (i.e., the average prevalence percentage of each independent element in README files). As expected, the proficient elements (Level 'C') are the less presented elements in the README files (i.e., the most presented element is 'Simple List Comprehension' at 3.52%).

However, by considering the usage of Python elements in each competency level group, the standard deviation of the basic group is 26.1, which can be interpreted that there are some elements that are highly presented more than other elements. As shown in Table 4.2, both elements 'Simple Attribute' and 'Simple Assignment' are presented at more than 80% while the rest elements are approximately presented at 65% and 35%. For the independent and proficient groups, the standard deviations are not considered as high when compared to the basic group (i.e., 3.92 and 1.19, respectively). So, I can interpret that there is no type of Python element from the independent and proficient groups highly presented in README files.

Summary 1: The most presented Python elements in README files are basic elements 'Simple Attribute' (87.22%) and 'Simple Assignment' (86.98%). The proficient elements are the less presented (i.e., maximum at 3.52%).

3.2 Python element usage in README files

A README file may comprise multiple competency levels of Python elements. The statistical calculation of average Python element usage in the README

Table 4.2. Prevalence of the top presented Python code elements from each competency level in the README files.

Competency Level	Python element	# README files	
Basic	Simple Attribute	1,413	(87.22%)
	Simple Assignment	1,409	(86.98%)
	From	1,054	(65.06%)
	Import	571	(35.25%)
	Simple List	564	(34.81%)
Independent	Inherited Class	291	(17.96%)
	Import with ‘as‘ extension	188	(11.60%)
	With	157	(9.69%)
	Nested Dictionary	142	(8.77%)
	Simple Class	137	(8.46%)
Proficient	Simple List Comprehension	57	(3.52%)
	Super Function	38	(2.35%)
	Generator Function (yield)	22	(1.36%)
	‘zip‘ call function	12	(0.74%)
	Generator Expression	12	(0.74%)

files is shown in Table 4.3. In a single README file, by considering the mean value, there are approximately 40 basic elements (Level ‘A’) and 3 independent elements (Level ‘B’) in the popular README file. The unpopular README file has approximately 22 basic elements and 2 independent elements. For the proficient elements (Level ‘C’), since the mean values are 0.20 (popular) and 0.16 (unpopular), it can be interpreted that these elements are presented in only a few README files regardless of the popularity.

If compare all README files together, by considering the median values, more than half of popular README files have 24 basic elements and 1 independent element. For the unpopular group, more than half of README files have 14 basic elements and 1 independent element. However, the number of basic elements in each popular README file is largely different since the standard deviation (SD) is 62.08. For instance, the library imgaug[6] has 389 basic elements in the README

[6] https://github.com/aleju/imgaug

file. Again, the proficient element is rarely found across all README files since the median is 0 and SD values are 1.01 (popular) and 0.74 (unpopular).

The column '% of elements' shows the number of elements from each competency level group in percentage. As shown in the Table 4.3, more than 90% of elements are basic level while independent elements are less than 10%. Moreover, there is no difference in the proportion of competency levels of Python elements between popular and unpopular groups.

Since the number of proficient elements is less than 1 in Table 4.3, I separately calculate the statistics average of Python element usage in the proficient README file only as shown in Table 4.4. Overall, the characteristic of element usage is the same with Table 4.3. However, there is an increase in element proportion. Not only increasing proficient elements (considering mean and median values) but also basic and independent elements. Again, there is no difference in the proportion of competency levels of Python elements between popular and unpopular groups.

For the statistical evaluation, the null hypothesis on whether 'the amount of Python elements from different competency levels are the same' is rejected for all popularity groups in both Table 4.3 and Table 4.4. So, I can interpret that the proportion of elements from each competency level is significant differences (i.e., p-value < 0.001) with a large effect size (Cliff's $\delta \geq 0.496$ in all groups).

Summary 2: Popular README files contain approximately 1.5 times more Python elements than unpopular README files. Especially for the proficient README files that contain approximately 2 times more Python elements.

3.3 Competency level of code snippets in README files

After grouping the README files based on the detected highest competency level of the code snippet, I can summarize the number of code snippets in each group of README files as follows.

Table 4.5 shows the statistics of the competency levels of Python code snippets from all popularity groups of README files. In a single README file,

Table 4.3. Summary statistics of Python elements from all README files separated by popularity.

Popularity	**Competency level of Python elements**	**Mean**	**Median**	**SD**	**% of elements**
Popular	Basic	40.24	24	62.08	92.65%
	Independent	3.02	1	5.69	6.69%
	Proficient	0.32	0	1.01	0.66%
Gap	Basic	26.34	17	28.61	94.27%
	Independent	1.57	1	3.04	4.98%
	Proficient	0.27	0	1.02	0.75%
Unpopular	Basic	22.23	14	29.41	91.72%
	Independent	1.93	1	4.33	7.85%
	Proficient	0.16	0	0.74	0.43%

Table 4.4. Summary statistics of Python elements from proficient README files separated by popularity.

Popularity	**Competency level of Python elements**	**Mean**	**Median**	**SD**	**% of elements**
Popular	Basic	67.73	37	115.34	88.07%
	Independent	5.02	3	5.81	7.53%
	Proficient	2.12	1	1.72	4.40%
Gap	Basic	46.26	29	46.44	90.50%
	Independent	2.39	1	4.14	4.07%
	Proficient	1.96	1	2.10	5.43%
Unpopular	Basic	51.17	34	47.32	85.72%
	Independent	5.07	3	7.03	8.54%
	Proficient	2.15	1	1.77	5.74%

by considering the mean value, there are approximately 3 basic code snippets and 1 independent code snippet in the popular README file. The unpopular README file has approximately 2 basic code snippets and 1 independent code snippet. For the proficient code snippet, since the mean values are 0.22 (popular) and 0.11 (unpopular), it can be interpreted that the proficient code snippet is rarely presented in only a few README files regardless of the popularity.

If compare all README files together, by considering the median and standard deviation (SD) values, more than half of README files have at least 1 basic code snippet and 1 independent code snippet regardless of the popularity with the low SD (less than 5.64). However, some of the popular README files contain a large number of code snippets. For instance, the library quantdsl[7] has 35 code snippets in the README file. Again, the proficient code snippet is rarely found across all README files since the median is 0 and SD values are 0.64 (popular) and 0.47 (unpopular).

The column '% of codes' shows the number of code snippets from each competency level group in percentage. As shown in the Table 4.5, more than 50% of code snippets are basic level while independent code snippets are less than 40%. Moreover, there is no difference in the proportion of competency levels of Python code snippets between popular and unpopular groups.

Since the number of proficient code snippets is less than one in Table 4.5, I separately calculate the statistics average of Python code snippets usage in the proficient README file only as shown in Table 4.6. Overall, the number of code snippets is not different with Table 4.5. However, there is a difference in code snippet proportion. The column '% of codes' shows that the popular README files have the same proportion between basic and proficient code snippets, while the unpopular README files tend to contain 2 times more proficient code snippets than basic code snippets. It can be interpreted that in the unpopular README files, developers prefer to present a few proficient code snippets more than adding code snippets with various competency levels.

Interestingly, by considering the competency level of README files in each popularity group as shown in Table 4.7. The result shows that approximately half of README files comprise at least one independent code snippet regardless

[7] https://github.com/johnbywater/quantdsl

of the popularity. However, the popular README files seem to contain 2 times more proficient code snippets than the unpopular README files.

Summary 3: In spite of the fact that README files mainly comprise basic Python elements. However, approximately half of README files comprise at least one independent code snippet. The popular proficient README files tend to have a balanced amount of different competency levels of code snippets while approximately half of the code snippets in the unpopular proficient README files are proficient level.

Table 4.5. Summary statistics of Python code snippets from all README files separated by popularity.

Popularity	**Competency level of Python codes**	**Mean**	**Median**	**SD**	**% of codes**
Popular	Basic	3.20	1	5.64	57.94%
	Independent	1.54	1	2.87	36.07%
	Proficient	0.22	0	0.64	5.99%
Gap	Basic	2.14	1	2.26	59.56%
	Independent	0.93	0	1.77	33.03%
	Proficient	0.20	0	0.85	7.41%
Unpopular	Basic	1.91	1	4.24	58.55%
	Independent	0.95	1	1.52	37.37%
	Proficient	0.11	0	0.47	4.08%

4 Discussion

Based on the results in this chapter, I made the following recommendations and highlighted the implications for maintainers and clients of the PyPI libraries, and software engineering researchers.

Maintainers of the PyPI libraries: The results reveal a statistical proportion of how developers present Python elements in code snippets across each competency level in the README files. Hence, to improve the quality of the

Table 4.6. Summary statistics of Python code snippets from proficient README files separated by popularity.

Popularity	**Competency level of Python codes**	**Mean**	**Median**	**SD**	**% of codes**
Popular	Basic	2.69	2	2.77	38.55%
	Independent	1.67	1	2.49	21.71%
	Proficient	1.49	1	0.94	39.74%
Gap	Basic	2.13	1	2.83	59.57%
	Independent	0.57	0	1.38	33.03%
	Proficient	1.47	1	1.88	7.40%
Unpopular	Basic	1.67	1	2.63	24.62%
	Independent	1.32	0	2.28	20.58%
	Proficient	1.46	1	0.97	54.80%

Table 4.7. Number of the README files from each competency level group separated by popularity.

Popularity	**Competency level of README files**	**# README files**	
Popular	Basic	112	(34.46%)
	Independent	164	(50.46%)
	Proficient	49	(15.08%)
	Sum	**325**	**(100%)**
Gap	Basic	69	(41.31%)
	Independent	75	(44.91%)
	Proficient	23	(13.78%)
	Sum	**167**	**(100%)**
Unpopular	Basic	484	(42.91%)
	Independent	560	(49.64%)
	Proficient	84	(7.45%)
	Sum	**1,128**	**(100%)**
	Total README files	**1,620**	

README files, I suggest maintainers present at least two to four Python code snippets (one for basic usage and the rest for additional examples). However, maintainers should mainly present basic code snippets more than proficient code snippets. As shown in the README file of PyPI library elasticmock[8], one code snippet is used to describe the basic usage of the library while others are used as examples of advanced usage. In the code snippet, maintainers should keep their code snippets easy to comprehend by mainly presenting basic elements approximately 88-92%, independent elements approximately 7-8%, and proficient elements approximately 0-5%. In this study, I did not consider that developers may provide natural language as a descriptive text to aid with code comprehension.

Clients of the PyPI libraries: The results reveal that most of the code snippets in README files of PyPI libraries are easy to comprehend. There are 40% probability that clients will encounter basic or independent README files and 10% for proficient README files. These README files have approximately two to four Python code snippets. There are 5% probability that clients will encounter proficient code snippets in the README files. However, in the proficient README files, 30% - 50% of code snippets will be proficient level. Hence, novice clients may struggle to fully comprehend proficient README files. In this work, I did not consider how natural language explanations along with code snippets aid with code comprehension.

Software engineering researchers: The statistical results show the proportion of Python elements and code snippets in the README files of PyPI libraries. It reveals that different usage of elements and code snippets can affect the popularity of the PyPI libraries. Hence, I suggest researchers investigate other effects of the usage of different competency levels of Python code elements in the README files.

5 Threats to Validity

In this section, I discuss the potential threats to validity as follows.

Internal validity: The main threat to internal validity is the approach to

[8] `https://github.com/vrcmarcos/elasticmock`

collecting Python code snippets from README files. This study relies on regular expression, so it may not be able to capture all kinds of code snippets on README files. To mitigate this, I randomly select ten README files, and then manually compare the number of Python code snippets in README files with the extracted result from my regular expression patterns. I iteratively checked and edited the pattern of regular expression until I was able to collect all of the Python code snippets.

External validity: There are two main threats to external validity in this study. The first threat is the generalization of the results. The study focuses solely on PyPI libraries, so, the results may not generalize to other ecosystems such as npm and Maven. However, I believe that the methodology in this study can be applied to other ecosystems. Hence, the immediate future work is to explore the competency level of code snippets from other programming languages or ecosystems. The second threat is the sample of libraries in our dataset. In this study, I created the dataset by selecting the libraries regardless of their popularity (e.g., stars, contributors, dependencies), as I would like to make the result generalized to all developers and clients regardless of their experience in Python programming.

Construct validity: In this study, I estimate the competency level of code snippets based on the highest level of Python elements inside. There are two main threats to constructing validity in this work. The first threat is that not all README files provide explanations along with code snippets. (i.e., library MIDITime[9] did provide but library Trip[10] did not) So, I decided to limit the scope of this study to code snippets analysis only. I acknowledge that well-written README files may sometimes contain explanations that help library users understand the usage of code snippets. This is indeed an interesting future direction to study. The second threat is the limitation of analyzing Python libraries based on topics in our study. Future work should identify other aspects (e.g., stars, contributors, and dependencies) to explore more benefits of analyzing the competency level of code snippets in the README files (e.g., increase popularity and accessibility).

[9] https://github.com/cirlabs/miditime

[10] https://github.com/littlecodersh/trip

6 Conclusion and Future Works

In this chapter, I conducted a quantitative study to investigate the prevalence of the competency level of Python code snippets in 1,620 README files from PyPI libraries. By adopting `pycefr`, a tool for detecting and calculating the competency level required to understand Python elements in the code snippets, I found that 45% - 50% of README files from PyPI libraries comprise independent Python code snippets while 34% - 43% comprise only basic Python code snippets. Even though the README files mainly comprise basic Python code elements. However, The popular proficient README files tend to have a balanced amount of different competency levels of code snippets while approximately half of the code snippets in the unpopular proficient README files are proficient level.

From the result, it shows that proficient Python code snippets may rarely be found in the README files. However, I believe that there are reasons or factors that affect developers to present proficient Python code snippets in their README files. Thus, this opens up an opportunity for future work to understand the reasons that affect how developers present different competency levels of code snippets in their README files.

5 | Usage of Python Code Snippets in different topics

1 Introduction

Nowadays, there is a large number of Open Source Software (OSS) projects hosted on various software repositories. For instance, GitHub as one of the largest software repositories currently host more than 330 million projects [1]. These projects can be categorized based on the domain or main functionality of the software [27] (e.g., programming language, library, database, application, and framework).

To utilize the software effectively, developers of the repositories often provide README files as a manual to use the software [83]. Liu et al. [62] encourages developers to provide README files with some code snippets in the sections about 'Usage' and 'Installation'. Antinyan et al. [7] suggests that code snippets should be readable with easy understandability, and reasonable complexity for code comprehension. However, some developers are more concerned about the performance when using their software [23]. So, they may present advanced code snippets as example usage in README files.

Motivated by this evidence, I hypothesized that there may be a correlation between the domains of software and the usage of code snippets. In this chapter, I created the dataset based on the 1,620 PyPI libraries from Chapter 4. Then I filtered the dataset based on the PyPI topics resulting in 1,598 README files of PyPI libraries as a case study. I then conducted a quantitative study with the

[1] https://github.com/about - (Accessed 23 May 2023)

goal of analyzing whether the topics of PyPI libraries affect the usage of Python code snippets in README files.

The results show that developers tend to create a similar proportion of all competency levels of README files in most topics but have a different proportion in certain topics. Proficient developers present a similar amount of independent and proficient Python elements in the code snippets.

The rest of this chapter is organized as follows. Section 2 describes my study design to analyze a correlation between the domains of software and the usage of code snippets. Section 3 describes the result in this chapter. Section 4 discusses the implication of this chapter. Section 5 describes threats to validity of this study. Finally, Section 6 summarizes the results and future works of this chapter.

2 Study Design

The goal of this study is to analyze whether the topics of PyPI libraries affect the usage of Python code snippets in README files. This section describes the research questions and study procedure to achieve the goal.

2.1 Research Questions

In this study, I define the following research questions (RQs).

- **RQ$_1$:** *To what extent do different topics of PyPI libraries have different proportions of competency levels of README files?* The motivation of this RQ is to investigate the proportions of competency level of README files from each topic of PyPI libraries. Each topic has different characteristics and purposes of usage, so, I hypothesize that some topics may comprise a higher proportion of proficient README files.

- **RQ$_2$:** *To what extent do different topics of PyPI libraries have different proportions of Python elements in each competency level group?* This RQ is a detailed analysis of the usage of Python elements in README files. The motivation of this RQ is that some topics of PyPI libraries may comprise the same proportions of competency level of README files but have different proportions of Python elements from the same level.

- **RQ$_3$:** *How do developers present proficient Python code snippets in proficient README files?* The motivation of this RQ is to investigate whether developers with high proficiency in Python programming tend to present proficient Python elements more than basic or independent elements.

- **RQ$_4$:** *How do developers present various types of code snippets in README files from different topics of PyPI libraries?* The motivation of this RQ is that developers not only present Python code snippets but also other types of code snippets in README files. I hypothesize that topics of libraries may affect how developers present different types of code snippets in each section of README files.

2.2 Data Preparation

In this chapter, I created the dataset based on the 1,620 PyPI libraries from Chapter 4. First, I extracted the topics of each library and then extracted different types of code snippets and their sections in the README files. The summaries of the dataset in this chapter are shown in Table 5.1, Table 5.2, and Table 5.3. The detail of dataset preparation is described as follows.

Step 1: Topic extraction from PyPI libraries. According to the PyPI classifier page[2], PyPI libraries can be classified into 24 topics (including the topic *Other*). The library can belong to single or multiple topics. Each topic of libraries may have different usage and proportions of code snippets due to the characteristics of the topics. First, I crawl the PyPI page of each library by using *requests*[3] to get page content in HTML format. I then apply *BeautifulSoup*[4] to find an HTML structure of the list of topics in the classifiers section. I filtered only the topics that comprise at least 30 libraries, which resulted in 1,598 libraries that belong to 23 topics remaining as shown in Table 5.1. The libraries that belong to the topic *Other* are excluded from this study. Table 5.2 shows the number of README files in each competency level group. Table 5.3 shows Python elements from each competency level group after filtering. Note that some README files

[2] https://pypi.org/classifiers

[3] https://pypi.org/project/requests

[4] https://pypi.org/project/beautifulsoup4

belong to multiple groups. In this case, I duplicated the README files and then separated them into the corresponding group.

Step 2: Code snippet types and sections extraction and categorization. In this step, I used regular expressions to extract the info_strings[5] (i.e. keyword used to specify the language of the code snippet), and section headings[6] from each README file while referencing the GitHub Flavored Markdown Spec (GFM)[7]. Since there is various type of code snippets, I classify them as follows.

- **Python source code**: Code snippet written using Python syntax.
- **Command script**: Code snippet that is typically executed as a command in a terminal and is usually written in bash, shell, etc.
- **Configuration**: Code snippet used for setting or customizing, typically written in a markup language format, e.g., XML, YAML, etc.
- **Source code**: Code snippet that is written in a programming language other than Python.
- **Markup language**: Code snippet that utilizes a text-encoding structure (tags) and is not intended for configuration, e.g., HTML, MD, etc.

For the section of each code snippet, I used the tool provided by the prior study [83] to categorize the sections into 8 categories (i.e., What, Why, How, When, Who, Reference, Contribution, and Other). However, sections that belong to the category Other are excluded from this study. Note that the tool may categorize some sections into multiple categories.

2.3 Data Analysis

To answer all RQs, I adopt the heatmap visualization as same as Islam et al. [44]. The heatmap uses colored-scale cells to show a two-dimensional matrix relationship between two data. I analyze the relationships between competency levels of

[5] https://github.github.com/gfm/#info-string

[6] https://github.github.com/gfm/#atx-heading

[7] https://github.github.com/gfm

Table 5.1. Prevalence of the README files in each topic.

Topic	**# README files**	
Software Development	932	(58.32%)
Scientific/Engineering	346	(21.65%)
Internet	289	(18.09%)
Utilities	264	(16.52%)
System	123	(7.70%)
Text Processing	78	(4.88%)
Database	49	(3.07%)
Multimedia	46	(2.88%)
Communications	37	(2.32%)
Office/Business	36	(2.25%)
Security	35	(2.19%)

code snippets in README files and topics of PyPI libraries by using a heatmap to visualize the relationships of the following pairs: (i) the relationship between competency levels of README files and topics of PyPI libraries, (ii) the relationship between topics of PyPI libraries and Python elements, (iii) the relationship between competency levels of README files and Python elements, and (iv) the relationship between types of code snippets and sections in README files. The heatmap reports the frequency counts in the percentage of each relationship that is reflected in the cells.

In order to statistically validate the relationships, I apply Pearson's chi-squared test (χ^2) [80]. I also investigate the effect size by using Cramér's V (ϕ') [21] which can be ranged from 0 (i.e., no association at all) to 1 (i.e., perfect association). The interpretation is shown in Table 5.4 according to [18]. The null hypothesis is that 'there is no association between the comparing pair'.

3 Results

From the quantitative study on the usage of code snippets in README files from each topic of the PyPI libraries, I can analyze each RQ as follows.

Table 5.2. Number of the filtered README files from 23 topics separated by popularity.

Popularity	**Competency level of README files**	**# README files**	
Popular	Basic	108	(33.75%)
	Independent	163	(50.94%)
	Proficient	49	(15.31%)
	Sum	**320**	**(100%)**
Gap	Basic	69	(41.82%)
	Independent	74	(44.85%)
	Proficient	22	(13.33%)
	Sum	**165**	**(100%)**
Unpopular	Basic	476	(42.76%)
	Independent	554	(49.78%)
	Proficient	83	(7.46%)
	Sum	**1,113**	**(100%)**
	Total README files	**1,598**	

3.1 RQ$_1$: To what extent do different topics of PyPI libraries have different proportions of competency levels of README files?

The heatmaps of Figure 5.1 show the percentage of README files from each competency level group that belongs to each topic separated by the popularity groups. By considering the percentage in the heatmaps, I describe the result interpretation as follows.

- Topic *Software Development* has the highest percentage of README files from all competency level groups, followed by topics *Scientific/Engineering* and *Internet*. Topic *Database* has the lowest percentage of README files in the popular group, while topic *Security* has the lowest percentage of README files in the unpopular group.

- In the popular group, developers mainly create basic README files in top-

Table 5.3. Prevalence of the top presented Python code elements from each competency level group in the README files from all topics.

Competency level	Python element	# README files	
Basic	Simple Attribute	1,395	(87.30%)
	Simple Assignment	1,390	(86.98%)
	From	1,039	(65.02%)
	Import	565	(35.36%)
	Simple List	555	(34.73%)
Independent	Inherited Class	287	(17.96%)
	Import with 'as' extension	187	(11.70%)
	With	156	(9.76%)
	Nested Dictionary	140	(8.76%)
	Simple Class	136	(8.51%)
Proficient	Simple List Comprehension	55	(3.44%)
	Super Function	38	(2.38%)
	Generator Function (yield)	22	(1.38%)
	'zip' call function	12	(0.75%)
	Generator Expression	12	(0.75%)

ics *Utilities* (10%) and *Text Processing* (7.6%). In contrast, topics *System* and *Communication* mainly comprise proficient README files instead. For the unpopular group, developers mainly create basic README files in topic *System* (8.6%). In contrast, topic *Scientific/Engineering* mainly comprises proficient README files instead.

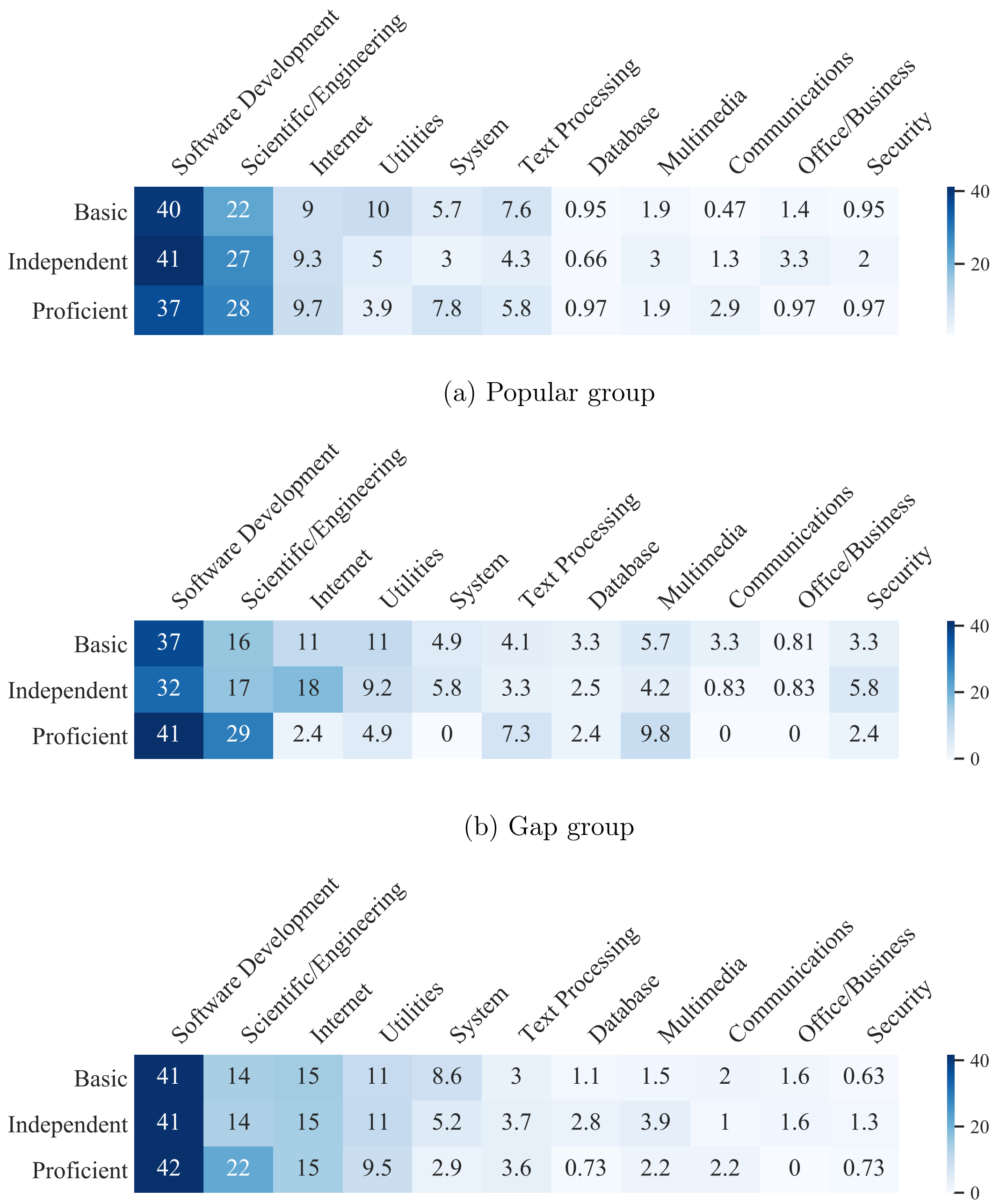
- To validate the relationship between the competency level groups of README files and topics, the statistical test is applied to each popularity group. Ac-

Table 5.4. Cramér's V effect size interpretation.

df*	Negligible	Small	Medium	Large
1	[0, 0.10)	[0.10, 0.30)	[0.30, 0.50)	0.50 or more
2	[0, 0.07)	[0.07, 0.21)	[0.21, 0.35)	0.35 or more
10	[0, 0.03)	[0.03, 0.10)	[0.10, 0.17)	0.17 or more

cording to Pearson's chi-squared test, the null hypothesis fails to reject in the popular and gap groups, i.e., the relationship between competency level groups and topics in these groups is not significant (p-value ≥ 0.3), while the effect size is small (Cramér's V values are 0.135 and 0.196, respectively). However, in the unpopular group, the null hypothesis is rejected, i.e., the relationship between competency level groups and topics in the unpopular group is significant (p-value < 0.05), while the effect size is small (Cramér's V 0.103). It can be interpreted that the topics of PyPI libraries are associated with how developers present different competency levels of code snippets in README files in the unpopular group but very slightly in the popular group.

RQ$_1$ Summary: In the popular group, developers tend to create a similar proportion of all competency levels of README files in most topics but have a different proportion in certain topics in the unpopular groups. For instance, the topic *System* mainly contains basic README files while the topic *Scientific/Engineering* mainly contains proficient README files instead.



(a) Popular group

(b) Gap group

(c) Unpopular group.

Figure 5.1. Heatmap of the relationship between competency level groups of README files and topics of PyPI libraries.

3.2 RQ$_2$: To what extent do different topics of PyPI libraries have different proportions of Python elements in each competency level group?

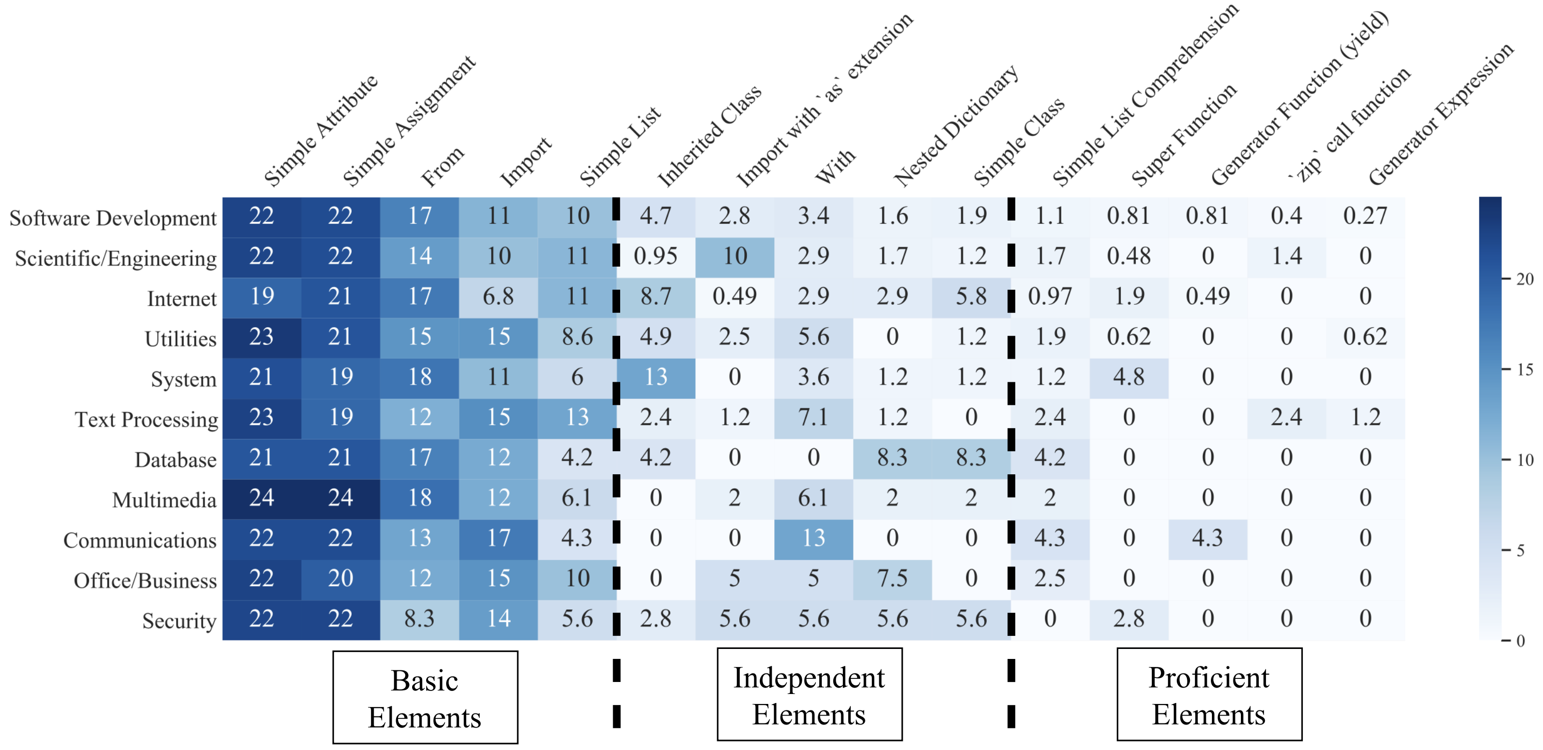
The heatmaps of Figure 5.2 show the percentage of Python elements usage in each topic of PyPI libraries separated by the popularity groups. In the heatmaps, Python elements are divided into three groups (i.e., Basic, Independent, and Proficient) by using the dash lines. To gain insight into how developers present each Python element in README files from different topics of PyPI libraries, I analyze the relationship between topics and Python elements as follows.

- In the basic group, elements 'Simple Attribute', 'Simple Assignment', and 'From' are the most generally presented elements across all topics. It can be implied that these three basic elements are commonly found regardless of the topics of libraries. In comparison with elements 'Import' and 'Simple List', the usage of these elements is slightly different in some topics. For instance, the element 'Import' is less found in topic *Internet* (6.8%) in the popular group and topics *Internet* (6.4%), *Database* (6.9%) and *Office/Business* (8%) in the unpopular group. In contrast, the element 'Simple List' is more found in topics *Text Processing* (13%), and *Office/Business* (10%) in the popular group and topic *Office/Business* (13%) in the unpopular group instead. This shows the first evidence that developers from different topics present different proportions of the same Python element in README files.

- For the independent group, the usage of elements between the popular and unpopular groups is more clearly different. In the popular group, some topics have a high usage of certain elements. For instance, element 'Inherited Class' in topic *System* (13%), element 'Import with 'as' extension' in topic *Scientific/Engineering* (10%), and element 'With' in topic *Communication* (13%). There are various elements that are not presented in the README file as indicated by zero (0%) in the heatmap. For the unpopular group, most elements are similarly presented in most topics except for topics *Scientific/Engineering* (7.8%), *Text Processing* (0%), *Multimedia* (0% and 9%), and *Communication* (0%). Interestingly, developers of topic *Security* tend to present more of the independent elements when compared with other topics in both popular and unpopular groups. These results enhance the evidence from the basic group that developers from different topics present different proportions of the same Python element in README files.

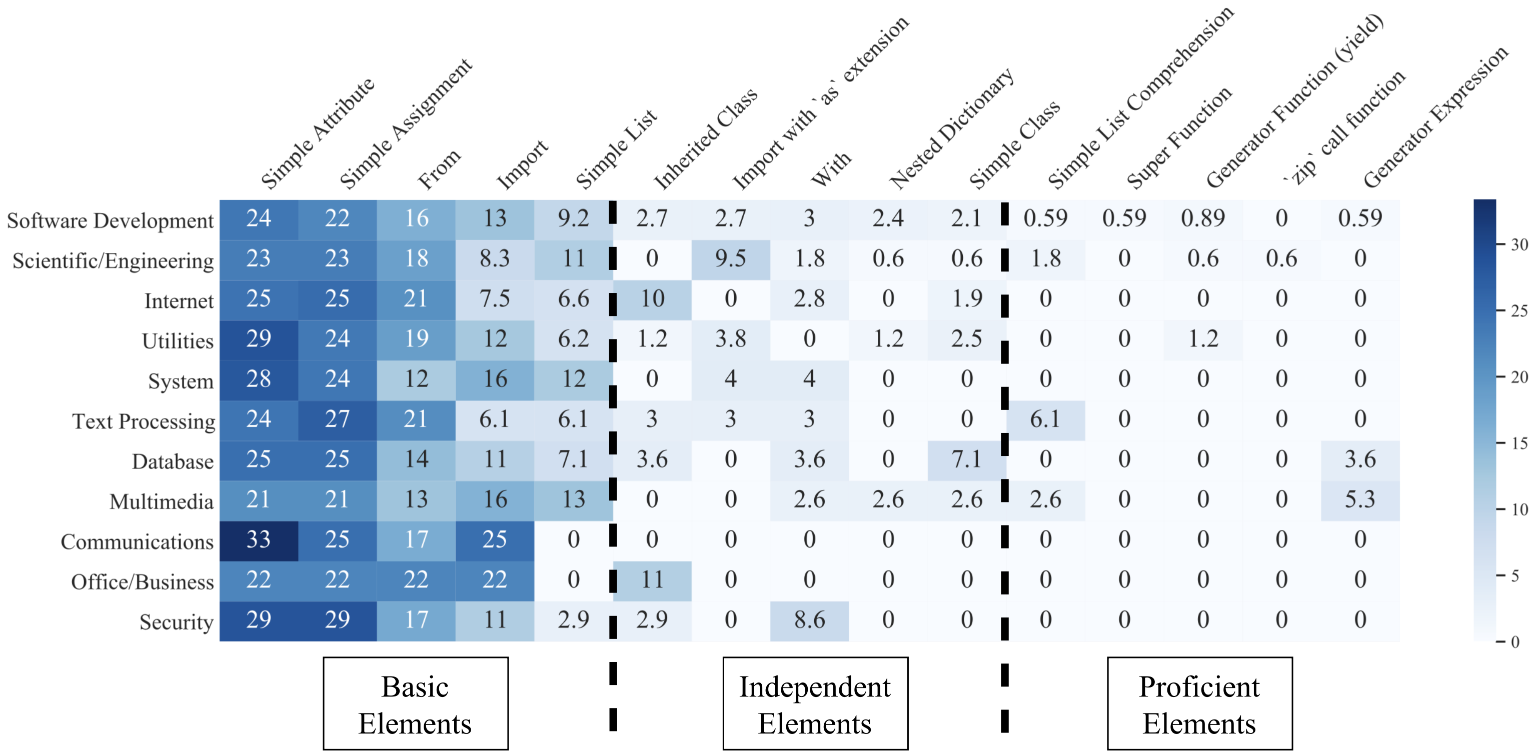
- In the proficient group, the prevalence of elements in each topic is quite similar to the independent group. Overall, the topics in the popular group have high usage of certain elements. For instance, the element 'Super Function' in topics *System* (4.8%), and *Security* (2.8%). However, topic *Communication* has high usage of elements 'Simple List Comprehension' and 'Generator Function (yield)' in both popular (4.3% and 4.3%, respectively) and unpopular groups (1.1% and 2.2%, respectively). Interestingly, in the unpopular group, it seems that README files from the topics *Office/Business* and *Security* have no proficient elements.

- To validate the relationship between Python elements in README files and topics, the statistical test is applied to each popularity group. According to Pearson's chi-squared test, the null hypothesis is rejected in the popular and unpopular groups, i.e., the relationship between topics and Python elements is significant in these groups (p-value < 0.001), while the effect size is medium and small (Cramér's V values are 0.114 and 0.070, respectively). However, in the gap group, the null hypothesis fails to reject, i.e., the relationship between topics and Python elements is not significant in the gap groups (p-value ≥ 0.2), while the effect size is medium (Cramér's V 0.131). It can be interpreted that the topics of PyPI libraries are associated with how developers present different Python elements in README files, especially between popular and unpopular groups.



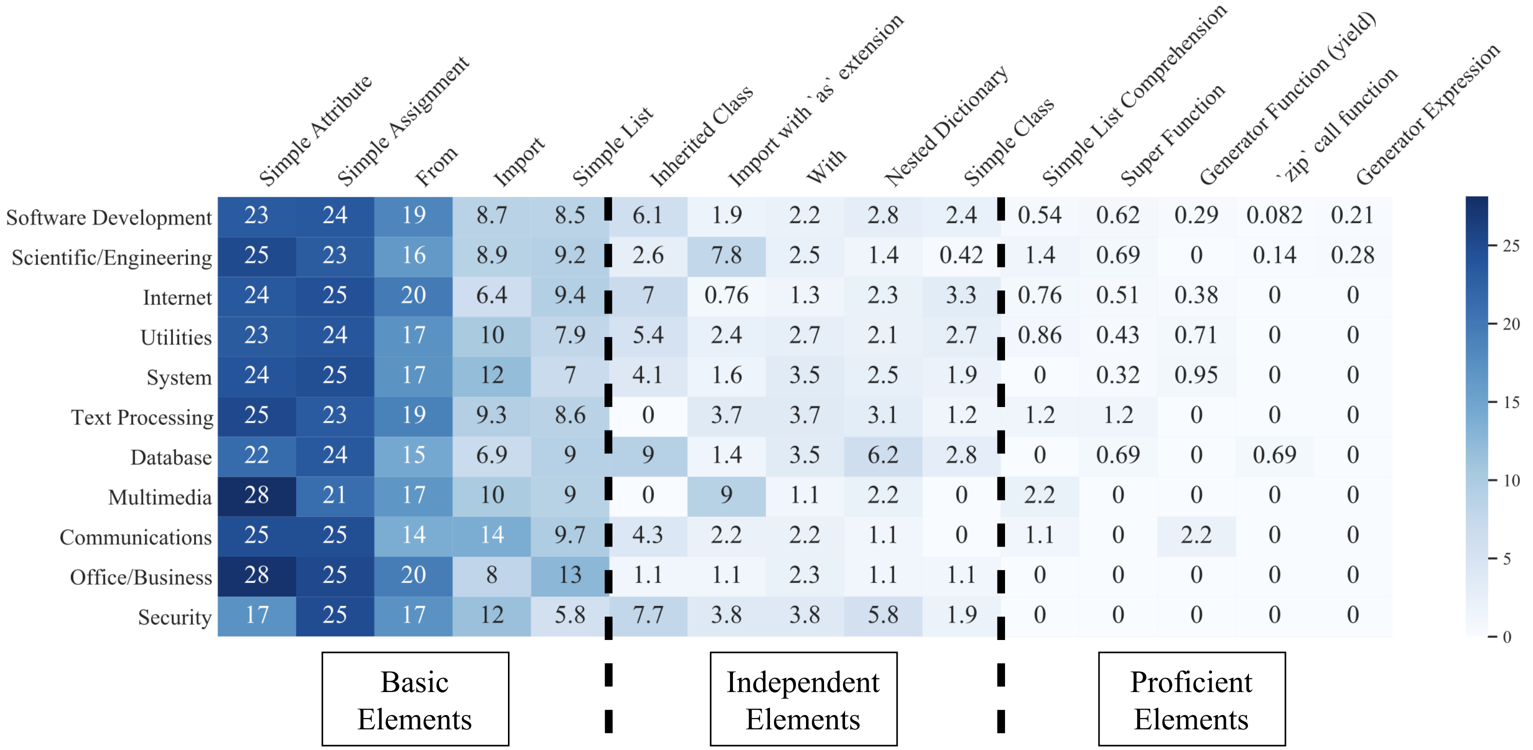
(a) Popular group

Figure 5.2. Heatmap of the relationship between topics and Python elements in each popularity group.



(b) Gap group

Figure 5.2. Heatmap of the relationship between topics and Python elements in each popularity group. (Cont.)



	Simple Attribute	Simple Assignment	From	Import	Simple List	Inherited Class	Import with `as` extension	With	Nested Dictionary	Simple Class	Simple List Comprehension	Super Function	Generator Function (yield)	`zip` call function	Generator Expression
Software Development	23	24	19	8.7	8.5	6.1	1.9	2.2	2.8	2.4	0.54	0.62	0.29	0.082	0.21
Scientific/Engineering	25	23	16	8.9	9.2	2.6	7.8	2.5	1.4	0.42	1.4	0.69	0	0.14	0.28
Internet	24	25	20	6.4	9.4	7	0.76	1.3	2.3	3.3	0.76	0.51	0.38	0	0
Utilities	23	24	17	10	7.9	5.4	2.4	2.7	2.1	2.7	0.86	0.43	0.71	0	0
System	24	25	17	12	7	4.1	1.6	3.5	2.5	1.9	0	0.32	0.95	0	0
Text Processing	25	23	19	9.3	8.6	0	3.7	3.7	3.1	1.2	1.2	1.2	0	0	0
Database	22	24	15	6.9	9	9	1.4	3.5	6.2	2.8	0	0.69	0	0.69	0
Multimedia	28	21	17	10	9	0	9	1.1	2.2	0	2.2	0	0	0	0
Communications	25	25	14	14	9.7	4.3	2.2	2.2	1.1	0	1.1	0	2.2	0	0
Office/Business	28	25	20	8	13	1.1	1.1	2.3	1.1	1.1	0	0	0	0	0
Security	17	25	17	12	5.8	7.7	3.8	3.8	5.8	1.9	0	0	0	0	0

(c) Unpopular group

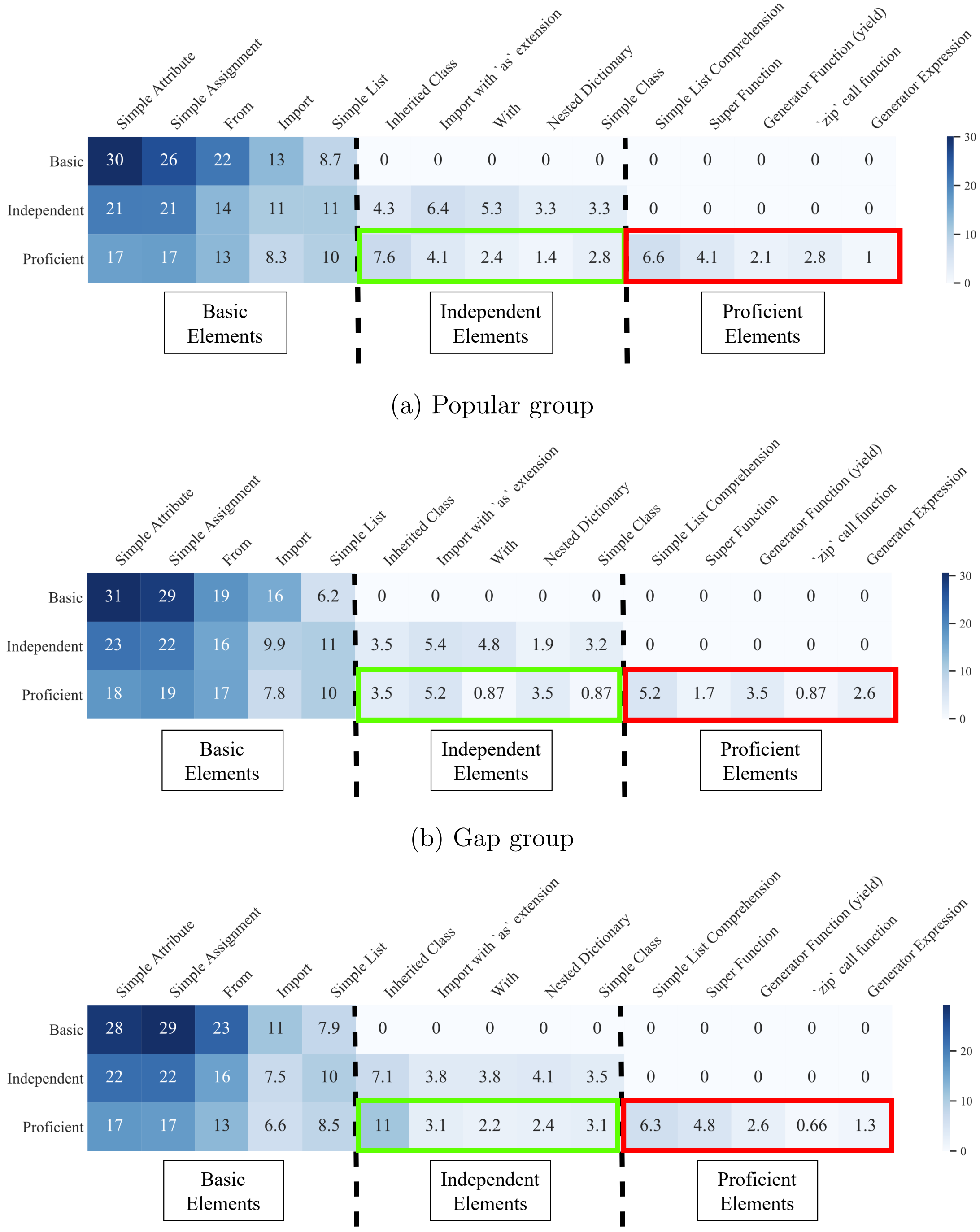
Figure 5.2. Heatmap of the relationship between topics and Python elements in each popularity group. (Cont.)

RQ$_2$ Summary: README files from some topics of PyPI libraries have high usage in certain Python elements, especially in the popular group. For instance, elements 'Inherited Class' and 'Super Function' in topic *System* and elements 'With' and 'Generator Function (yield)' in topic *Communication*. Interestingly, the topics *Office/Business* and *Security* have no presence of proficient elements in the unpopular group.

3.3 RQ$_3$: How do developers present proficient Python code snippets in proficient README files?

By considering only the proficient group in Figure 5.3, it seems that developers tend to present a similar percentage of independent elements (i.e., elements highlighted by a green box) and proficient elements (i.e., elements highlighted by a red box) but excluding element 'Inherited Class'. Since execution performance is one of the most important factors that developers are concerned with when using libraries [23]. So, I conducted a manual investigation on how proficient elements are presented in README files. As a result, I found that elements 'Simple List Comprehension', 'Generator Function', and 'Generator Expression' were usually written in a Pythonic way. I believe that one of the reasons that developers tend to present Pythonic elements is to achieve better execution performance [54].

Since it is obvious that README files in the basic group never contain elements from higher levels. So, I apply Pearson's chi-squared to test the association between the percentages of independent elements (i.e., green box) and proficient elements (i.e., red box) in the proficient group instead. The null hypothesis fails to reject in all popularity groups, i.e., the relationship between the independent elements and proficient elements in the proficient group is not significant (p-value ≥ 0.05), while the effect size is small, large, and small (Cramér's V values are 0.227, 0.519 and 0.285) for popular, gap and unpopular group, respectively. It can be interpreted that the percentage of proficient elements is slightly lower than independent elements in README files regardless of the popularity.



(a) Popular group

(b) Gap group

(c) Unpopular group.

Figure 5.3. Heatmap of the relationship between competency level group of README files and Python elements in each popularity group.

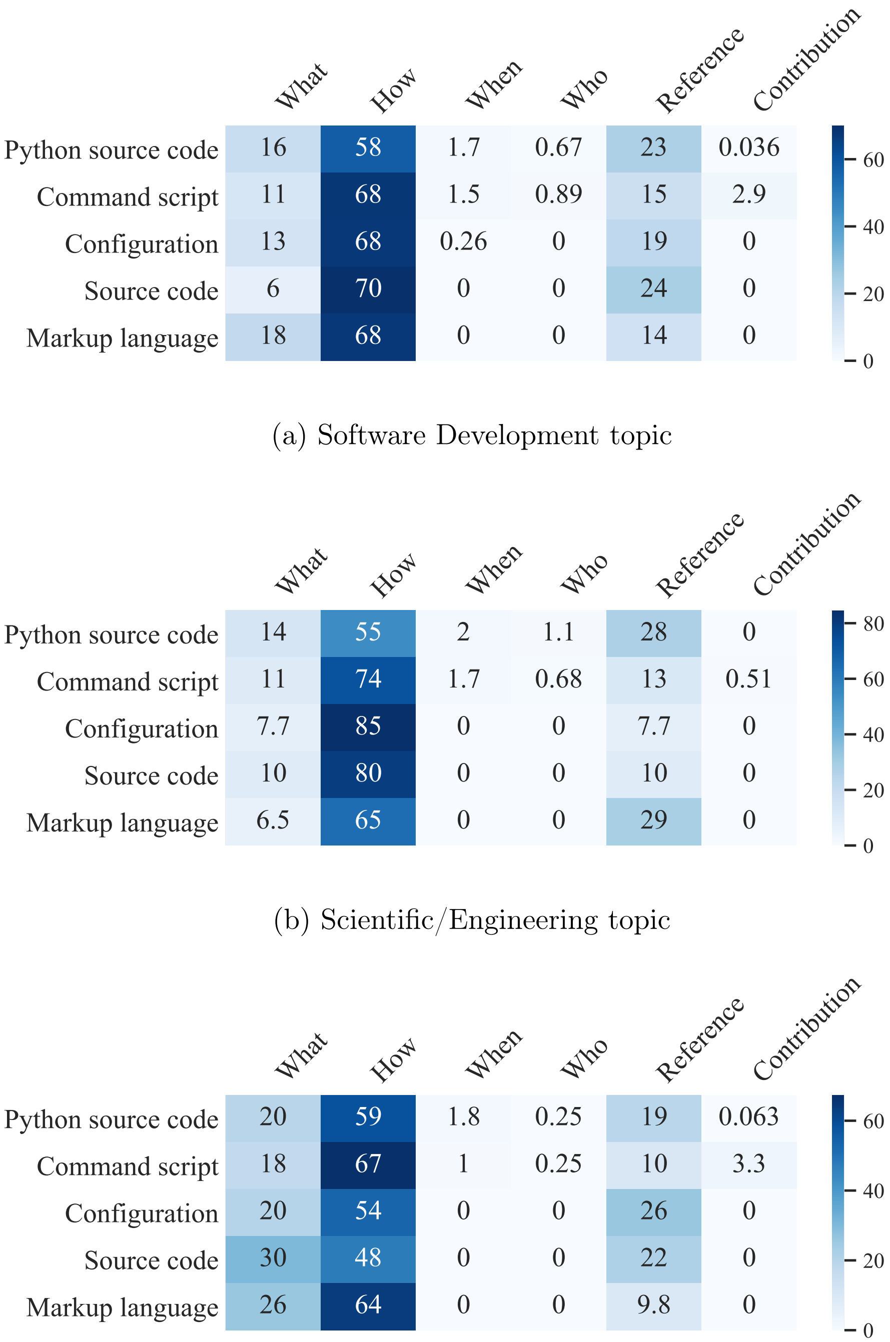
RQ$_3$ Summary: Proficient developers present a similar amount of independent and proficient Python elements in the code snippets. The elements that are usually written in Pythonic way are ‘Simple List Comprehension’, ‘Generator Function’, and ‘Generator Expression’.

3.4 RQ$_4$: How do developers present various types of code snippets in README files from different topics of PyPI libraries?

The heatmaps of Figure 5.4 show the percentage of usage of five types of code snippets in six sections of README files (Section Why is excluded from the heatmaps due to no presence of code snippet in all topics). However, some topics have less or none of some type of code snippets and/or less or none of code snippets in some sections. In this case, I exclude those types of code snippets and or sections from the heatmaps. By considering the percentage in the heatmaps, I describe the result interpretation as follows.

- Overall, all types of code snippets are mainly presented in the section Why followed by sections What and Reference. Topics *Software Development*, *Scientific/Engineering*, *System*, *Text Processing* and *Office/Business* have a high proportion of code snippets in the section Reference, while topics *Internet*, *Database* and *Multimedia* have a high proportion of code snippets in the section What. The topics *Utilities*, *Communication* and *Security* have a similar proportion of code snippets in both sections What and Reference.

- The content in the How section of the README files is about the usage and installation of the PyPI libraries. The Command script is the most presented type of code snippet in the How section except for topics *Software Development*, *Scientific/Engineering*, *Utilities*, *Office/Business* and *Security*. The developers of these excluded topics tend to present other types of code snippets instead. For instance, the type Markup language in the topic *Utilities* (80%), and the type Configuration in the topics *Scientific/Engineering* (85%)and *Office/Business* (75%).

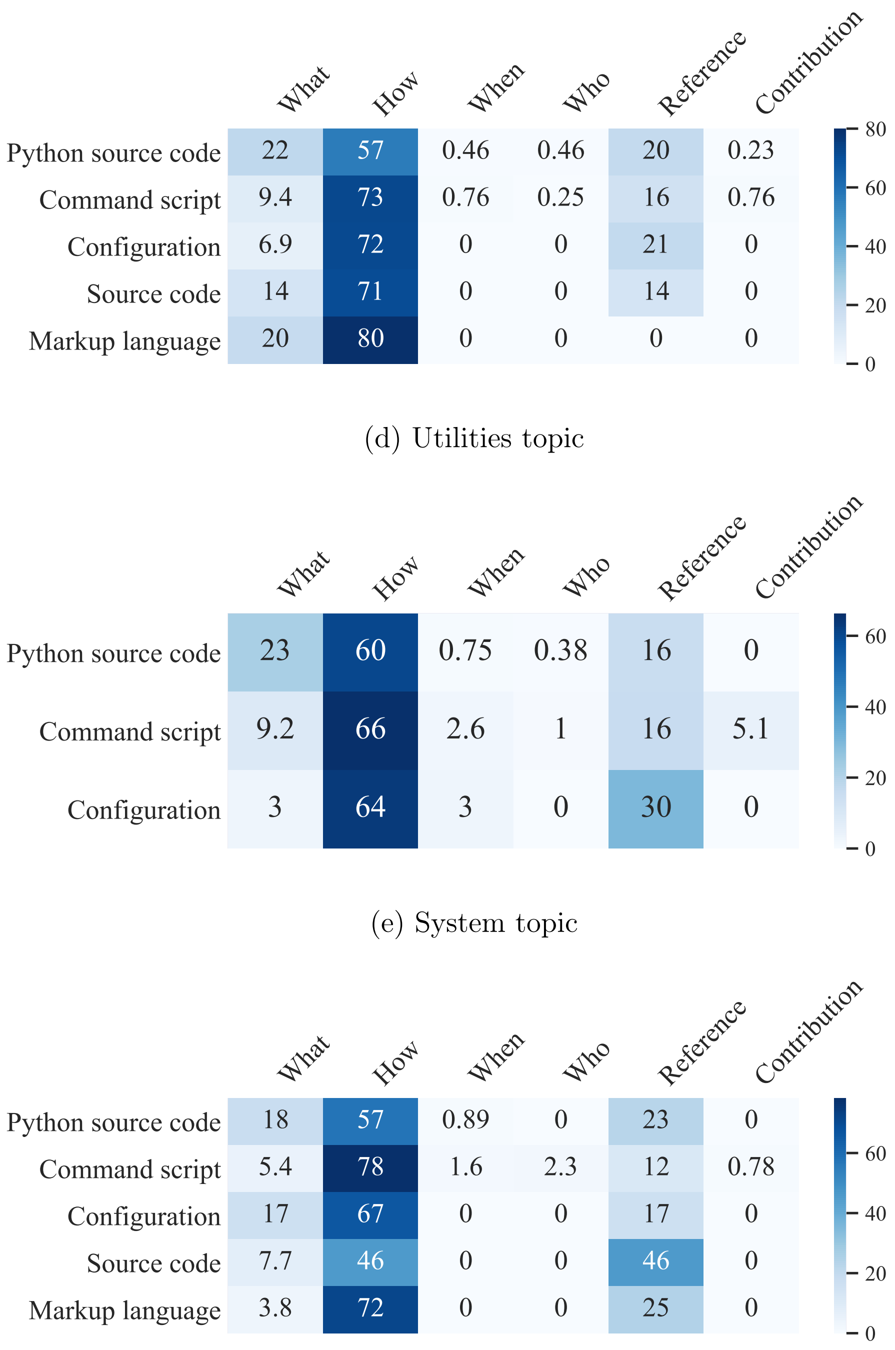


(a) Software Development topic

(b) Scientific/Engineering topic

(c) Internet topic

Figure 5.4. Heatmap of the relationship between types of code snippets and sections in README files from each topic.

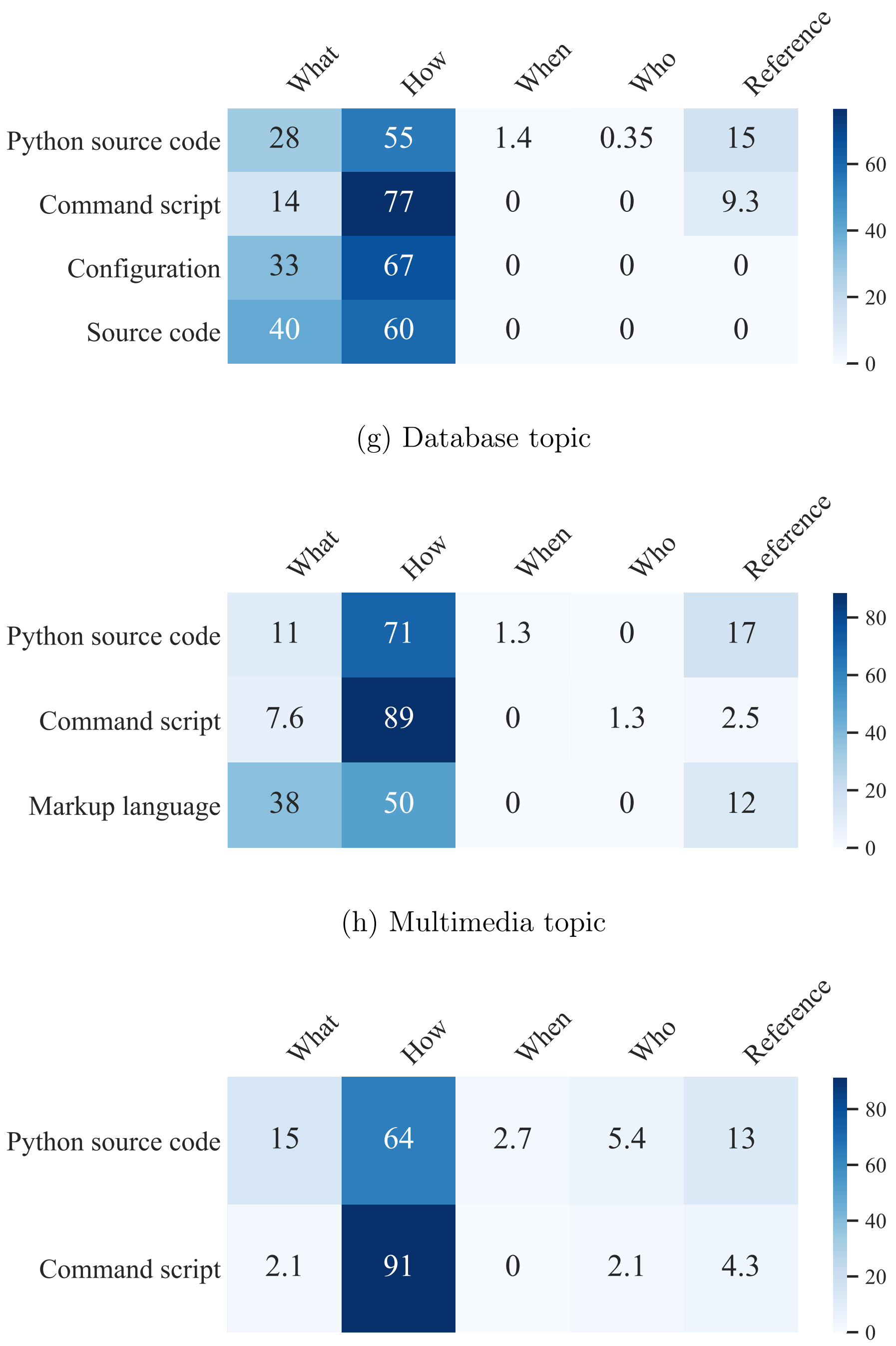


(d) Utilities topic

(e) System topic

(f) Text Processing topic

Figure 5.4. Heatmap of the relationship between types of code snippets and sections in README files from each topic. (Cont.)

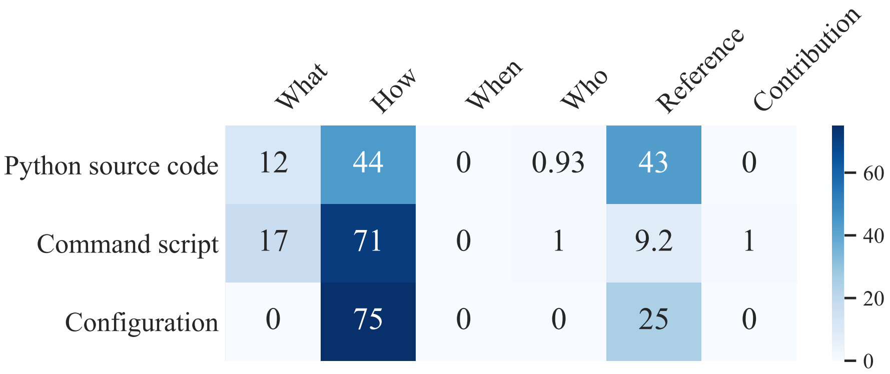


(g) Database topic

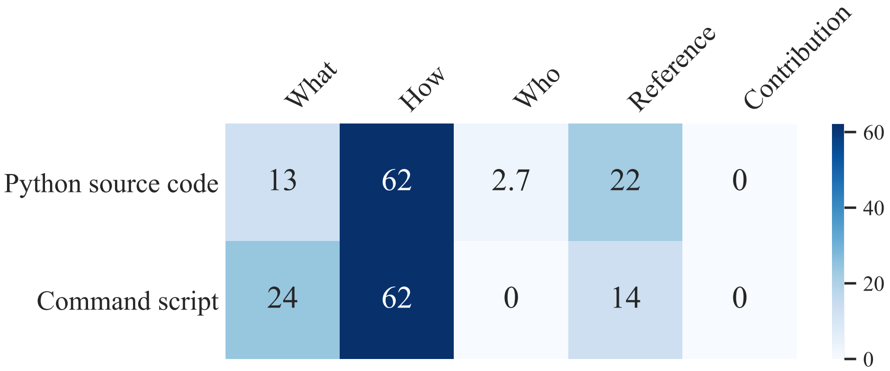
(h) Multimedia topic

(i) Communications topic

Figure 5.4. Heatmap of the relationship between types of code snippets and sections in README files from each topic. (Cont.)



(j) Office/Business topic



(k) Security topic

Figure 5.4. Heatmap of the relationship between types of code snippets and sections in README files from each topic. (Cont.)

- The goal of this RQ is to investigate the proportion of the usage of each type of code snippet in different sections of README files, not the relationship. So, the statistical test to validate the relationship between the types of code snippets and sections in README files is not necessary for this RQ.

RQ$_4$ Summary: All types of code snippets are mainly presented in the How section of README files, especially the type Command script. Developers from each topic of PyPI libraries tend to present each type of code snippet in different proportions and sections.

4 Discussion

Based on the results of this study, there is a correlation between the usage of Python code snippets and the topic of PyPI libraries. In this section, I discuss implications and recommendations as follows.

Maintainers of the PyPI libraries: The results show that topics of PyPI libraries affect the competency level of Python code snippets and the location of each type of code snippet in the README files. Hence, to improve the quality of README files, maintainers are suggested to adopt my results as a guideline to add code snippets in the README files based on the topic of PyPI libraries. For instance, maintainers of the topic *Scientific/Engineering* should consider adding more proficient elements, e.g., 'Simple List Comprehension' and "'.zip.' call function' because these elements are written in Pythonic which provide more execution performance. However, if the maintainers are not familiar with Pythonic code, some elements can be replaced with lower competency elements, e.g., the element 'Simple List Comprehension' (proficient level) can be replaced by 'Lambda' (independent level). In the How section of the README files, the maintainers of the topic *Scientific/Engineering* are suggested to provide the type of code snippets for configuration. As shown in the README file of PyPI library lisflood-utilities[8], it provides an example of a metadata configuration code snippet for using the library.

Clients of the PyPI libraries: The results show that README files from some topics of PyPI libraries comprise a higher number of proficient code snippets, especially the topic *Scientific/Engineering*. Hence, novice clients may struggle to comprehend some README files from these topics. Additionally, README files from some topics of PyPI libraries have high usage in certain Python elements.

[8] https://github.com/ec-jrc/lisflood-utilities

For instance, elements ‘Inherited Class’ and ‘Super Function’ in the topic *System*. Hence, the clients are suggested to use my results as a recommendation to study certain elements that are usually found in the README files from topics that are related to their field.

Software engineering researchers: The results indicate that the topics of PyPI libraries have an association with the usage of code snippets in README files. It reveals that some topics have different proportions of competency levels of Python elements. Additionally, each topic has a different usage of each type of code snippet in each section of README files. Hence, I encourage researchers to adopt my methodology to investigate other characteristics (e.g., contributors, and dependencies) that may be affected by the usage of code snippets in the README files from different software domains.

5 Threats to Validity

In this section, I describe the threats that may affect this study.

Internal validity: Threats to internal validity concern bias in this study. The main internal threat that may affect the result is the skewness of the number of PyPI libraries in my dataset. As shown in Table 5.1, topic *Software Development* takes 58.32% of the dataset while there are 6 out of 11 topics that take less than 10%. To mitigate this threat, I calculate the correlations between the competency level of Python code snippets and the topic of PyPI libraries by converting the number of libraries into percentages.

External validity: Threats to external validity concern the generalizability of the results. In this study, I define a case study by using Python code snippets in README files of PyPI libraries from 11 topics. Additionally, in this study, I apply `pycefr` to calculate the competency level of Python code snippets. So, the results may not fully generalize to all ecosystems. However, my methodology can apply to some ecosystems or programming languages that share the same characteristics as PyPI or Python, e.g., JavaScript and PHP. Future work may adopt the algorithm of `pycefr` to calculate the competency level of different programming languages and then follow the rest of my methodology.

Construct validity: Threats to construct validity concern the suitability of my correlation analysis. In each RQ, I scope my analysis to the correlation between Python code snippets in README files and the topic of PyPI libraries only. There is the possibility that other artifacts or components in PyPI libraries may affect my correlation analysis. Thus, this can be considered for future work.

6 Conclusion and Future Works

In this chapter, I conducted a quantitative study to analyze the correlation between the topic of PyPI libraries and the usage of code snippets in 1,598 README files. The analysis is based on the interpretation of the heatmap which visualizes the relationship between the Python elements in code snippets, the competency level of README files, the topics of PyPI libraries, types of code snippets, and sections in README files.

The results show that developers tend to create a similar proportion of all competency levels of README files in most topics but have a different proportion in certain topics. For instance, the topic *System* mainly contains basic README files while the topic *Scientific/Engineering* mainly contains proficient README files instead. Proficient developers present a similar amount of independent and proficient Python elements in the code snippets. The elements that are usually written in Pythonic way are 'Simple List Comprehension', 'Generator Function', and 'Generator Expression'. Additionally, All types of code snippets are mainly presented in the How section of README files, especially the type Command script. Developers from each topic of PyPI libraries tend to present each type of code snippet in different proportions and sections.

From the result, it seems that the domain of software is one of the reasons that affect the usage of code snippets in README files. Future work can extend my methodology to broader categories of software or other ecosystems. I believe that there are other factors that also affected by the usage of code snippets in README files.

6 Conclusion

As a leading open-source version control platform, GitHub hosts over 330 million repositories. It functions as a social coding platform, featuring tools for software projects, and attracting contributions. Encouraging active engagement from newcomers is crucial for project success, often achieved through comprehensive documentation.

README files are meta-documents recommended by GitHub for quick access to essential information. It plays an important role as a welcome page of software repositories. The well-written README files should provide the necessary information for clients to understand the purpose and usage of the repositories. The popular repositories often comprise some code snippets as visual content to demonstrate how software should be used.

To utilize the software effectively, clients should be able to comprehend all code snippets in the README files. However, clients sometimes encounter advanced elements that might prevent them from comprehending code snippets. One of the reasons is that some developers are concerned about the execution performance of their software, so, they prefer to present proficient code snippets in the README files instead.

In this study, I conducted an empirical study to analyze the correlation between the competency level of code snippets in README files and the domain of software. By using PyPI libraries as a case study, I designed the study in this thesis with the goals of investigating (i) the prevalence of different competency levels of code snippets in README files, and (ii) whether topics of PyPI libraries affect how developers present code snippets in README files.

For the first goal, I conducted a quantitative study to investigate the prevalence of the competency level of Python code snippets in 1,620 README files from PyPI libraries. By adopting `pycefr`, a tool for detecting and calculating the competency level required to understand Python elements in the code snippets, I found that 45% - 50% of README files from PyPI libraries comprise independent Python code snippets while 34% - 43% comprise only basic Python code snippets. Even though the README files mainly comprise basic Python code elements. However, The popular proficient README files tend to have a balanced amount of different competency levels of code snippets while approximately half of the code snippets in the unpopular proficient README files are proficient level.

For the second goal, I conducted a quantitative study to analyze the correlation between the topic of PyPI libraries and the usage of code snippets in 1,598 README files. The analysis is based on the interpretation of the heatmap which visualizes the relationship between the Python elements in code snippets, the competency level of README files, and the topics of PyPI libraries. The results show that developers tend to create a similar proportion of all competency levels of README files in most topics but have a different proportion in certain topics. Proficient developers present a similar amount of independent and proficient Python elements in the code snippets. Developers from each topic of PyPI libraries tend to present each type of code snippet in different proportions and sections.

In summary, the results of this thesis highlight how developers present different competency levels of Python code snippets based on the topics of PyPI libraries.

1 Implications

Based on the results of this thesis, I made the following recommendations and highlighted the implications for maintainers and clients of the PyPI libraries, and also software engineering researchers.

Developers of the PyPI Libraries: to improve the quality of the README files, I suggest maintainers present at least two to four Python code snippets (one for basic usage and the rest for additional examples). However, maintainers should mainly present basic code snippets more than proficient code snippets. As shown in the README file of PyPI library elasticmock[1], one code snippet is used to describe the basic usage of the library while others are used as examples of advanced usage. In the code snippet, maintainers should keep their code snippets easy to comprehend by mainly presenting basic elements approximately 88-92%, independent elements approximately 7-8%, and proficient elements approximately 0-5%.

For example, maintainers of the topic *Scientific/Engineering* should consider adding more proficient elements, e.g., 'Simple List Comprehension' and ".zip.' call function' because these elements are written in Pythonic which provide more execution performance. However, if the maintainers are not familiar with Pythonic code, some elements can be replaced with lower competency elements, e.g., the element 'Simple List Comprehension' (proficient level) can be replaced by 'Lambda' (independent level). In the How section of the README files, the maintainers of the topic *Scientific/Engineering* are suggested to provide the type of code snippets for configuration. As shown in the README file of PyPI library lisflood-utilities[2], it provides an example of a metadata configuration code snippet for using the library.

Clients of the PyPI Libraries: There are 40% probability that clients will encounter basic or independent README files and 10% for proficient README files. These README files have approximately two to four Python code snippets. There are 5% probability that clients will encounter proficient code snippets in the README files. However, in the proficient README files, 30% - 50% of code snippets will be proficient level. Hence, novice clients may struggle to fully comprehend proficient README files.

Additionally, README files from some topics of PyPI libraries have high usage in certain Python elements. For instance, elements 'Inherited Class' and 'Super Function' in the topic *System*. Hence, the clients are suggested to use my results as a recommendation to study certain elements that are usually found in the README files from topics that are related to their field.

Software Engineering Researchers: The results reveal that different us-

[1] https://github.com/vrcmarcos/elasticmock

[2] https://github.com/ec-jrc/lisflood-utilities

age of elements and code snippets can affect the popularity of the PyPI libraries, and also some topics have different proportions of competency levels of Python elements. The results indicate that the topics of PyPI libraries have an association with the usage of code snippets in README files. Additionally, each topic has a different usage of each type of code snippet in each section of README files. Hence, I encourage researchers to adopt my methodology to investigate other characteristics (e.g., contributors, and dependencies) that may be affected by the usage of code snippets in the README files from different software domains.

2 Opportunities for Future Research

In this thesis, I studied how developers present code snippets in README files from the PyPI libraries. There are additional possible aspects that can be done in order to assist developers in creating quality README files. The outlines of the research opportunities for the immediate future are listed as follows.

- **Integrating with other elements in README files.** README files can contain both texture descriptions and non-textual elements (e.g., images, badges, and links). Future works should consider analyzing these elements with the code snippets in order to extend the recommendation to create quality README files.

- **Additional factors in the repositories.** Apart from the domain of software, there might be other factors that affect the contents of README files (e.g., stars, download counts, repository ages, and number of contributors). Future works should apply these factors to analyze the content that must be considered when creating quality README files.

- **Expand studying to other software ecosystems or programming languages.** Although I conducted the study in this thesis by using PyPi libraries as a case study. I believe that my methodology can be a guideline for conducting a related study in other software ecosystems or programming languages. Future works should apply additional factors to study whether the contents of README files are affected by different categories of software.

References

[1] Karan Aggarwal, Abram Hindle, and Eleni Stroulia. Co-evolution of project documentation and popularity within github. In Proceedings of IEEE/ACM Mining Software Repositories Conference (MSR), pages 360–363, 2014.

[2] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In Proceedings of the 41st International Conference on Software Engineering, ICSE '19, page 1199–1210. IEEE Press, 2019. doi: 10.1109/ICSE.2019.00122.

[3] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C Shepherd. Software documentation: the practitioners' perspective. In 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pages 590–601. IEEE, 2020.

[4] Carol V. Alexandru, José J. Merchante, Sebastiano Panichella, Sebastian Proksch, Harald C. Gall, and Gregorio Robles. On the usage of pythonic idioms. In Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, page 1–11, 2018.

[5] Doaa Altarawy, Hossameldin Shahin, Ayat Mohammed, and Na Meng. Lascad : Language-agnostic software categorization and similar application detection. Journal of Systems and Software (JSS), 142:21–34, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.04.018.

[6] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In 2010 IEEE International Conference on Software Maintenance, pages 1–10, 2010. doi: 10.1109/ICSM.2010.5609747.

[7] Vard Antinyan, Miroslaw Staron, and Anna Börjesson Sandberg. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. Empirical Software Engineering, 22:3057–3087, 2017.

[8] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, page 157–166. Association for Computing Machinery, 2010. ISBN 9781605587912. doi: 10.1145/1882291.1882316.

[9] Sebastian Baltes and Stephan Diehl. Usage and attribution of stack overflow code snippets in github projects. Empirical Software Engineering (EMSE), 24(3): 1259–1295, 2019. ISSN 1382-3256. doi: 10.1007/s10664-018-9650-5.

[10] Sebastian Baltes, Richard Kiefer, and Stephan Diehl. Attribution required: Stack overflow code snippets in github projects. In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pages 161–163, 2017. doi: 10.1109/ICSE-C.2017.99.

[11] Andrew Begel, Jan Bosch, and Margaret-Anne Storey. Social networking meets software development: Perspectives from github, msdn, stack exchange, and topcoder. IEEE Software, 30(1):52–66, 2013. doi: 10.1109/MS.2013.13.

[12] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 334–344, 2016. doi: 10.1109/ICSME.2016.31.

[13] John Businge, Alexander Serebrenik, and Mark van den Brand. Analyzing the eclipse api usage: Putting the developer in the loop. In 2013 17th European Conference on Software Maintenance and Reengineering, pages 37–46, 2013. doi: 10.1109/CSMR.2013.14.

[14] Yingkui Cao, Yanzhen Zou, Yuxiang Luo, Bing Xie, and Junfeng Zhao. Toward accurate link between code and software documentation. Science China Information Sciences, 61, 2018. ISSN 1869-1919. doi: 10.1007/s11432-017-9402-3.

[15] Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. Do people prefer "natural" code?, 2019.

[16] Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. Do programmers prefer predictable expressions in code? Cognitive Science, 44 (12):e12921, 2020.

[17] Stephen Cass. Top programming languages 2021: Python dominates as the de facto platform for new technologies. https://spectrum.ieee.org/top-programming-languages-2021, 2022.

[18] Jacob Cohen. Statistical Power Analysis for the Behavioral Sciences. Routledge, 1988.

[19] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pages 275–284, 2014. doi: 10.1109/SCAM.2014.14.

[20] Valerio Cosentino, Javier L. Cánovas Izquierdo, and Jordi Cabot. A systematic mapping study of software development with github. IEEE Access, 5:7173–7192, 2017. doi: 10.1109/ACCESS.2017.2682323.

[21] Harald Cramér. Mathematical Methods of Statistics. Princeton University Press, 1946.

[22] Barthélémy Dagenais and Martin P. Robillard. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, page 127–136, 2010.

[23] Fernando López de la Mora and Sarah Nadi. Which library should i use? a metric-based comparison of software libraries. In Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, page 37–40, 2018.

[24] Alexandre Decan, Tom Mens, Maëlick Claes, and Philippe Grosjean. When github meets cran: An analysis of inter-repository package dependency problems. In Proceedings of International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 493–504, 2016.

[25] Andrea Di Sorbo, Sebastiano Panichella, Carol V. Alexandru, Junji Shimagaki, Corrado A. Visaggio, Gerardo Canfora, and Harald C. Gall. What would users change in my app? summarizing app reviews for recommending software changes. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, page 499–510, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950299.

[26] O. Elazhary, M. Storey, N. Ernst, and A. Zaidman. Do as i do, not as i say: Do contribution guidelines match the github contribution process? In 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 286–290. IEEE Computer Society, 2019. doi: 10.1109/ICSME.2019.00043.

[27] Javier Escobar-Avila, Mario Linares-Vásquez, and Sonia Haiduc. Unsupervised software categorization using bytecode. In 2015 IEEE 23rd International Conference on Program Comprehension, pages 229–239, 2015. doi: 10.1109/ICPC.2015.33.

[28] Yuanrui Fan, Xin Xia, David Lo, Ahmed E. Hassan, and Shanping Li. What makes a popular academic AI repository? Empirical Software Engineering (EMSE), 26 (1), 2021.

[29] Eric Firing. Tutorial part i: Python language elements. https://currents.soest.hawaii.edu/ocn_data_analysis/python_tutorial.html, 2022. (Accessed on 19/03/2023).

[30] Shlok Gilda. Source code classification using neural networks. In 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE), pages 1–6, 2017. doi: 10.1109/JCSSE.2017.8025917.

[31] Mark Grechanik, Kevin M. Conroy, and Katharina A. Probst. Finding relevant applications for prototyping. In Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007), pages 12–12, 2007. doi: 10.1109/MSR.2007.9.

[32] Gillian J. Greene and Bernd Fischer. Cvexplorer: Identifying candidate developers by mining and exploring their open source contributions. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering,

page 804–809, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338455. doi: 10.1145/2970276.2970285.

[33] Piyush Gupta, Nikita Mehrotra, and Rahul Purandare. Jcoffee: Using compiler feedback to make partial code snippets compilable. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 810–813, 2020. doi: 10.1109/ICSME46990.2020.00099.

[34] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In Proceedings of the 2010 17th Working Conference on Reverse Engineering, WCRE '10, page 35–44, USA, 2010. IEEE Computer Society. ISBN 9780769541235. doi: 10.1109/WCRE.2010.13.

[35] Junxiao Han, Shuiguang Deng, Xin Xia, Dongjing Wang, and Jianwei Yin. Characterization and prediction of popular projects on github. In 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), volume 1, pages 21–26, 2019. doi: 10.1109/COMPSAC.2019.00013.

[36] Foyzul Hassan and Xiaoyin Wang. Mining readme files to support automatic building of java projects in software repositories. In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pages 277–279, 2017. doi: 10.1109/ICSE-C.2017.114.

[37] Hideaki Hata, Taiki Todo, Saya Onoue, and Kenichi Matsumoto. Characteristics of sustainable oss projects: A theoretical and empirical study. In 2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering, pages 15–21, 2015. doi: 10.1109/CHASE.2015.9.

[38] Hideaki Hata, Christoph Treude, Raula Gaikovina Kula, and Takashi Ishio. 9.6 million links in source code comments: Purpose, evolution, and decay. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 1211–1221. IEEE, 2019.

[39] Claudia Hauff and Georgios Gousios. Matching github developer profiles to job advertisements. In Proceedings of IEEE/ACM Mining Software Repositories Conference (MSR), pages 362–366, 2015.

[40] B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic. A domain-specific software architecture for adaptive intelligent systems. IEEE Transactions on Software Engineering (TSE), 21(4):288–301, 1995. doi: 10.1109/32.385968.

[41] E. Horton and C. Parnin. Gistable: Evaluating the executability of python code snippets on github. In Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 217–227, 2018.

[42] GitHub Inc. Basic writing and formatting syntax - github docs. https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax, 2022. (Accessed on 12/05/2023).

[43] GitHub Inc. About readmes - github docs. https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes, 2022. (Accessed on 12/05/2023).

[44] Syful Islam, Raula Gaikovina Kula, Christoph Treude, Bodin Chinthanet, Takashi Ishio, and Kenichi Matsumoto. Contrasting third-party package management user experience. In 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 664–668, 2021. doi: 10.1109/ICSME52107.2021.00077.

[45] Siyuan Jiang and Collin McMillan. Towards automatic generation of short summaries of commits. In 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pages 320–323, 2017. doi: 10.1109/ICPC.2017.12.

[46] S. Kawaguchi, P.K. Garg, M. Matsushita, and K. Inoue. Mudablue: an automatic categorization system for open source repositories. In 11th Asia-Pacific Software Engineering Conference, pages 184–193, 2004. doi: 10.1109/APSEC.2004.69.

[47] Aidan Kelley and Daniel Garijo. A framework for creating knowledge graphs of scientific software metadata. Quantitative Science Studies, 2(4):1423–1446, 2021. ISSN 2641-3337. doi: 10.1162/qss_a_00167.

[48] Miika Koskela, Inka Simola, and Kostas Stefanidis. Open source software recommendations using github. In International Conference on Theory and Practice of Digital Libraries (TPDL), 2018.

[49] Rrezarta Krasniqi, Siyuan Jiang, and Collin McMillan. Tracelab components for generating extractive summaries of user stories. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 658–658, 2017. doi: 10.1109/ICSME.2017.86.

[50] William H Kruskal and W Allen Wallis. Use of Ranks in One-Criterion Variance Analysis. Journal of the American Statistical Association (JASA), 47(260):583–621, 1952. ISSN 0162-1459.

[51] Senthil Kumar Kumarasamy Mani, Rose Catherine, Vibha Sinha, and Avinava Dubey. Ausum: Approach for unsupervised bug report summarization. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012, page 11, 11 2012. doi: 10.1145/2393596.2393607.

[52] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. Selecting third-party libraries: The practitioners' perspective. In Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), page 245–256, 2020.

[53] Alexander LeClair, Zachary Eberhart, and Collin McMillan. Adapting neural text classification for improved software categorization. In Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 461–472, 2018.

[54] Pattara Leelaprute, Bodin Chinthanet, Supatsara Wattanakriengkrai, Raula Gaikovina Kula, and Takashi Ishio. Does coding in pythonic zen peak performance? preliminary experiments of nine pythonic idioms at scale. In Proceedings of International Conference on Program Comprehension (ICPC), pages 575–579, 2022.

[55] T.C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: the state of the practice. IEEE Software, 20(6):35–39, 2003. doi: 10.1109/MS.2003.1241364.

[56] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Nicholas Kraft. Automatically documenting unit test cases. In 2016 IEEE

International Conference on Software Testing, Verification and Validation (ICST), pages 341–352, 2016. doi: 10.1109/ICST.2016.30.

[57] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Aiding comprehension of unit test cases and test suites with stereotype-based tagging. In Proceedings of the 26th Conference on Program Comprehension, ICPC '18, page 52–63, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357142. doi: 10.1145/3196321.3196339.

[58] Mario Linares-Vásquez, Collin Mcmillan, Denys Poshyvanyk, and Mark Grechanik. On using machine learning to automatically classify software applications into domain categories. Empirical Software Engineering (EMSE), 19(3): 582–618, 2014. ISSN 1382-3256. doi: 10.1007/s10664-012-9230-z.

[59] Mario Linares-Vásquez, Boyang Li, Christopher Vendome, and Denys Poshyvanyk. Documenting database usages and schema constraints in database-centric applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, page 270–281, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931072.

[60] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. Changescribe: A tool for automatically generating commit messages. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 2, pages 709–712, 2015. doi: 10.1109/ICSE.2015.229.

[61] Mario Linares-Vásquez, Andrew Holtzhauer, and Denys Poshyvanyk. On automatically detecting similar android apps. In 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pages 1–10, 2016. doi: 10.1109/ICPC.2016.7503721.

[62] Yuyang Liu, Ehsan Noei, and Kelly Lyons. How readme files are structured in open source java projects. Information and Software Technology (IST), 148: 106924, 2022.

[63] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. Modelling the 'hurried' bug report reading process to summarize bug reports. In 2012 28th IEEE International Conference on Software Maintenance (ICSM), pages 430–439, 2012. doi: 10.1109/ICSM.2012.6405303.

[64] Allen Mao, Daniel Garijo, and Shobeir Fakhraei. Somef: A framework for capturing scientific software metadata from its documentation. In 2019 IEEE International Conference on Big Data (Big Data), pages 3032–3037, 2019. doi: 10.1109/BigData47090.2019.9006447.

[65] Ami Marowka. On parallel software engineering education using python. Education and Information Technologies (EAIT), 23(1):357–372, 2018.

[66] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014, page 279–290, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328791. doi: 10.1145/2597008.2597149.

[67] Paul W. McBurney and Collin McMillan. Automatic source code summarization of context for java methods. IEEE Transactions on Software Engineering (TSE), 42(2):103–119, 2016. doi: 10.1109/TSE.2015.2465386.

[68] Paul W. McBurney, Cheng Liu, Collin McMillan, and Tim Weninger. Improving topic model source code summarization. In Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014, page 291–294, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328791. doi: 10.1145/2597008.2597793.

[69] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. Detecting similar software applications. In 2012 34th International Conference on Software Engineering (ICSE), pages 364–374, 2012. doi: 10.1109/ICSE.2012.6227178.

[70] Qing Mi, Jacky Keung, Yan Xiao, Solomon Mensah, and Yujin Gao. Improving code readability classification using convolutional neural networks. Information and Software Technology, 104:60–71, 2018. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.07.006.

[71] Samim Mirhosseini and Chris Parnin. Docable: Evaluating the executability of software tutorials. In Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), page 375–385, 2020.

[72] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In 2013 21st International Conference on Program Comprehension (ICPC), pages 23–32, 2013. doi: 10.1109/ICPC.2013.6613830.

[73] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014) Proceedings. November 16–21, 2014 Hong Kong, China, volume 16-21, pages 484–495, New York, 2014. Association for Computing Machinery, Inc. ISBN 978-1-4503-3056-5.

[74] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Arena: An approach for the automated generation of release notes. IEEE Transactions on Software Engineering (TSE), 43(2):106–127, 2017. doi: 10.1109/TSE.2016.2591536.

[75] Kawser Wazed Nafi, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. A universal cross language software similarity detector for open source software categorization. Journal of Systems and Software (JSS), 162:110491, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110491.

[76] Council of Europe. Common european framework of reference for languages (cefr). https://www.coe.int/en/web/common-european-framework-reference-languages, 2022.

[77] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. Evaluating code readability and legibility: An examination of human-centric studies. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 348–359, 2020. doi: 10.1109/ICSME46990.2020.00041.

[78] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, page 547–558, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884847.

[79] Yunrim Park and Carlos Jensen. Beyond pretty pictures: Examining the benefits of code visualization for open source newcomers. In 2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, pages 3–10, 2009. doi: 10.1109/VISSOF.2009.5336433.

[80] Karl Pearson. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 50(302):157–175, 1900.

[81] Clifton Phua, Damminda Alahakoon, and Vincent Lee. Minority report in fraud detection: Classification of skewed data. ACM SIGKDD Explor. Newsl., 6(1): 50–59, jun 2004. ISSN 1931-0145. doi: 10.1145/1007730.1007738. URL https://doi.org/10.1145/1007730.1007738.

[82] Roxana Lisette Quintanilla Portugal and Julio Cesar Sampaio do Prado Leite. Extracting requirements patterns from software repositories. In 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW), pages 304–307, 2016.

[83] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. Categorizing the content of github readme files. Empirical Software Engineering (EMSE), 24(3):1296–1327, 2019.

[84] Huilian Qiu, Yucen Li, Susmita Padala, Anita Sarma, and Bogdan Vasilescu. The signals that potential contributors look for when choosing open-source projects. Proceedings of the ACM on Human-Computer Interaction, 3:1–29, 2019.

[85] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. Automatic summarization of bug reports. IEEE Transactions on Software Engineering (TSE), 40(4):366–380, 2014. doi: 10.1109/TSE.2013.2297712.

[86] Brittany Reid, Christoph Treude, and Markus Wagner. Optimising the fit of stack overflow code snippets into existing code. In Proceedings of Genetic and Evolutionary Computation Conference (GECCO), page 1945–1953, 2020.

[87] Martin P. Robillard. What makes apis hard to learn? answers from developers. IEEE software, 26(6):27–34, 2009.

[88] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, et al. On-demand developer documentation. In 2017 IEEE International conference on software maintenance and evolution (ICSME), pages 479–483. IEEE, 2017.

[89] G. Robles, R. Kula, C. Ragkhitwetsagul, T. Sakulniwat, K. Matsumoto, and J. M. Gonzalez-Barahona. pycefr: Python competency level through code analysis. In Proceedings of International Conference on Program Comprehension (ICPC), pages 173–177, 2022.

[90] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, page 390–401, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568247.

[91] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices. In Annual meeting of the Southern Association for Institutional Research, 2006.

[92] Simone Scalabrino, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In Proceedings of International Conference on Program Comprehension (ICPC), pages 1–10, 2016.

[93] Abhishek Sharma, Ferdian Thung, Pavneet Singh Kochhar, Agus Sulistya, and David Lo. Cataloging github repositories. In Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (EASE), page 314–319, 2017.

[94] Dan Sholler, Igor Steinmacher, Denae Ford, Mara Averick, Mike Hoye, and Greg Wilson. Ten simple rules for helping newcomers become contributors to open projects. PLOS Computational Biology, 15:e1007296, 2019.

[95] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods.

In Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, ASE '10, page 43–52, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450301169. doi: 10.1145/1858996.1859006.

[96] Kai Tian, Meghan Revelle, and Denys Poshyvanyk. Using latent dirichlet allocation for automatic categorization of software. In 2009 6th IEEE International Working Conference on Mining Software Repositories, pages 163–166, 2009. doi: 10.1109/MSR.2009.5069496.

[97] Christoph Treude, Justin Middleton, and Thushari Atapattu. Beyond accuracy: Assessing software documentation quality. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, page 1509–1512, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3417045.

[98] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem. In Proceedings of International Conference on Software Engineering (ICSE), pages 511–522, 2018.

[99] Gias Uddin and Martin P Robillard. How api documentation fails. Ieee software, 32(4):68–75, 2015.

[100] Tao Wang, Huaimin Wang, Gang Yin, Charles X. Ling, Xiang Li, and Peng Zou. Mining software profile across multiple repositories for hierarchical categorization. In 2013 IEEE International Conference on Software Maintenance, pages 240–249, 2013. doi: 10.1109/ICSM.2013.35.

[101] Supatsara Wattanakriengkrai, Bodin Chinthanet, Hideaki Hata, Raula Kula, Christoph Treude, Jin L.C. Guo, and Kenichi Matsumoto. Github repositories with links to academic papers: Public access, traceability, and evolution. Journal of Systems and Software, 183:111117, 10 2021. doi: 10.1016/j.jss.2021.111117.

[102] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pages 53–64. IEEE, 2019.

[103] Eliane S. Wiese, Anna N. Rafferty, and Armando Fox. Linking code readability, structure, and comprehension among novices: It's complicated. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), pages 84–94, 2019. doi: 10.1109/ICSE-SEET.2019.00017.

[104] Meng Yan, Xin Xia, Xiaohong Zhang, Dan Yang, and Ling Xu. Automating aggregation for software quality modeling. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 529–533, 2017. doi: 10.1109/ICSME.2017.30.

[105] Annie T. T. Ying and Martin P. Robillard. Code fragment summarization. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, page 655–658, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322379. doi: 10.1145/2491411.2494587.

[106] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. Detecting similar repositories on github. In Proceedings of International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 13–23, 2017.

[107] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, and Liming Zhu. Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms. In Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2022.

[108] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, Genoa, page 318–343, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 9783642030123. doi: 10.1007/978-3-642-03013-0_15.

[109] Jing Zhou and Robert J. Walker. Api deprecation: A retrospective analysis and detection method for code examples on the web. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, page 266–277, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950298.