Exploring the Correspondence Between Types of Documentation for Application Programming Interfaces

Deeksha Arya

School of Computer Science
McGill University
Montreal, Quebec, Canada

November 2019

A thesis submitted to McGill University in partial
fulfillment of the requirements of the degree of
Master of Science

Abstract

Documentation of software programming languages and their APIs exist in many forms, whether as official reference documentation, user-created blog posts or other textual and visual mediums. Prior research has suggested that developers often switch between different types of documentation while learning a new API, with a tendency to alternate between reference and tutorial-like documentation. Further, documentation creation is an effort-intensive process that often leads to repeated information across different documentation types, generating a risk of information inconsistency. This thesis explores the relationship between instructional and API reference documentation of three libraries on the topics: regular expressions, URL connectivity and file input/output in two programming languages, Java and Python. Our investigation discovers that about half the sentences in the instructional documentations studied describe API-related information, such as syntax, behaviour, usage and performance of the API, that is expected to be found in the reference documentation. We also study the extent of information reuse across the documentation types, focusing on sentences in instructional documentation that are *exact*, *manipulated* and *replaceable* matches of those in reference documentation. We elicit four information reuse patterns based on our observations and discover a total of 38 instances of these patterns in the studied instructional documentations. We propose techniques to assist automation of each reuse pattern to reduce documentation creation efforts, inform documentation design and promote information consistency. We assess the impact of automation of these reuse patterns on the current documentation and determine that 15 instances of these patterns will not result in any information loss, the remaining affected with varying levels of modification. This work is a first step towards understanding the nature of information reuse

across different documentation types. Future work can use our observations to improve documentation artifacts and automate the instructional documentation creation process.

Resumé

La documentation des langages de programmation et de leurs interfaces de programmation (API, de l'anglais Application Programming Interface) existe sous plusieurs formes, qu'il s'agisse de références officielles, d'articles de blogs crées par les utilisateurs, ou d'autres supports textuels et visuels. Des recherches antérieures ont suggéré que les développeurs apprenant une nouvelle API passent souvent d'un type de documentation à l'autre, avec une tendance à alterner entre les références officielles et la documentation de type tutoriel. De plus, la création de documentation est un processus exigeant beaucoup d'efforts qui mène souvent à une répétition de l'information entre différents types de documentation, ce qui crée le risque d'insérer de l'information contradictoire. Cette thèse explore la relation entre la documentation instructive et la documentation de référence des API de trois bibliothèques de domaines différents, les expressions régulières, les connections URL et les entrées-sorties du système de fichiers, dans deux langages de programmation, Java et Python. Notre étude a révélé qu'environ la moitié des phrases de la documentation instructive étudiée décrivent des renseignements relatifs à l'API que l'on s'attendrait à retrouver dans la documentation de référence. Nous étudions également l'étendue de la réutilisation de l'information entre les types de documentation, en nous concentrant sur les phrases de la documentation instructive qui sont des copies *exactes*, *manipulées* ou *remplaçables* par rapport à celles de la documentation de référence. Nous obtenons quatre modèles de réutilisation de l'information basés sur nos observations et découvrons un total de 38 réalisations de ces modèles dans la documentation instructive étudiée. Nous proposons des techniques aidant l'automatisation de chaque modèle de réutilisation pour réduire les efforts liés à la création de documentation, informer la conception de la documentation et promouvoir la

cohérence de l'information. Nous évaluons l'impact de l'automatisation de ces modèles de réutilisation sur la documentation actuelle et déterminons que 15 des 38 réalisations de ces modèles n'entraîneront pas de perte d'information, les autres étant modifiées à différents niveaux. Cette thèse est une première étape vers la compréhension de la nature de la réutilisation de l'information à travers différents types de documentation. Les travaux futurs peuvent utiliser nos observations pour améliorer les artefacts documentaires, et automatiser le processus de création de documentation instructive.

Acknowledgements

A number of people have contributed either directly or indirectly towards the completion of this thesis, and I express my deepest gratitude to them.

First and foremost, I am extremely thankful to my supervisors, Prof. Jin Guo and Prof. Martin P. Robillard for their continuous support and guidance. Prof. Guo has been a source of ever-present encouragement since I stepped into research. An inspiration in the academic field and otherwise, I only hope that some day I am able to emulate the mentorship she provided me, to help another hopeful student truly enjoy their work and research like I did. I consider myself fortunate to have had the privilege of working with Prof. Robillard, whose patient guidance and excellent advice has sculpted my academic work and writing. Every meeting with him challenged me to work hard and smart, and so in every step, I saw myself improving and learning. I hope, through this work, I have been able to live up to the expectations of both my wonderful supervisors.

I would like to thank Prof. Jörg Kienzle for reviewing and providing constructive feedback on this work. I would also like to thank Prof. Jinghui Cheng, one of my first collaborators, for his insight and advice which has greatly helped in honing my research skills. My peers in the Knowledge Enhanced Software Connectivity (KESEC) and Software Evolution and Research Group (SWEVO) have become more friends than labmates. I am truly grateful to Mathieu for his help in editing the French abstract of this thesis and his useful inputs. I always turn to him for a fresh perspective, be it on work or otherwise, and he never fails to provide thought-provoking comments. Thank you to Alex and Cheryl for all their help and valuable feedback. Our brain-storming, idea-generating discussions, deep thoughts on

what research is and random every day banter made the lab a great place to learn and enjoy. Thank you to my friends here at McGill (special shout out to Anand, Lalita, Arushi and Ayush) with whom hard work days ended in relaxed evenings, and whose motivation kept me persistent in this thesis journey.

My heartfelt thanks to my now-husband, Vignesh, who would drop everything to take over my other responsibilities, so I could focus and dedicate my entire time and effort on this thesis. He always provides the encouragement I need, and reminds me to keep my chin up even through difficult times. To receive such support is the greatest thing I could ask for.

Most important of all, thanks are simply insufficient for my mother and father. Their unconditional support and steadfast belief in me and my abilities is what keeps me going. As my first and foremost teachers and my sources of inspiration, I hope I have made them proud. My sister, Maitri, has been, is and will always be my pillar of strength, and for that I will be ever-grateful. As always, she is my friend, my mom, my dad, my sister and my brother all in one. To all three of them and my loving grandparents, lots of love.

Copyright

This work focuses on the documentation of APIs in Java and Python. All screenshots of the documentation are protected by the following copyrights and are not owned by the author of this work. Further, neither of the corporations are affiliated or associated with this work.

We do not reiterate these copyrights for each reference made, however we state here that they are implied for these references in the entirety of this thesis.

Contents

List of Figures

List of Tables

1

Introduction

Imagine having to build a sofa from different available parts, without having the exact instructions on how to go about it. The time and effort it would take, (not to mention the frustration), is the reason that a user manual has become an integral part of assembly, from furniture to machinery, to make the construction as smooth and easy as possible for a potential user. This is the case in software development as well.

With the innumerable software development toolkits and the ever-increasing set of programming languages available, these technologies must be accompanied by appropriate documentation for developers and users to understand the functionalities the system provides and how to use them efficiently during software development and maintenance. Based on its target users and its end goal, documentation may be presented in many forms, from requirements documentation to user guides, from design documents to white papers. For example, comments in function headers generally inform users of accepted parameters and return types, whereas tutorials describe available functionality of a language component and how to use it correctly to achieve optimal performance. It is important that the statements made about a software component are consistent and when applicable, complementary, despite being presented with different perspectives and intentions.

Despite the availability of documentation, there exist gaps between the information that it provides and that which users seek [33, 40]. Previous research has discovered that some of the issues developers have with documentation include insufficient examples or a lack of how-to-use descriptions. User studies have shown that scenario-based instruction is an

integral resource for developers, who often switch between reference documentation and cook-book like documentation during software development [25, 26].

This thesis work is motivated by the fact that content in different documentation types about the same application programming interface (API) are related and developers tend to switch between them based on their information needs during the development process. We explored the relationship between API reference documentation and instructional documentation of two programming languages and their APIs, namely `Java` and `Python`. Specifically, we performed a case study on the reuse of information from API reference documentation to instructional documentation, where instructional documentation comprises of either general tutorials or in-depth How-Tos on specific concepts and libraries.

We performed an inter-document correlation exploration by beginning with sentences in instructional documentation and attempting to trace them back to the corresponding API reference. We chose to do this in order to assess the reusability of API reference documentation in supplementary instructional documentation. Further, we explored the extent of automatability of instructional documentation generation based on this reuse. We followed an inductive technique, formalizing characteristics and patterns and deriving conclusions based on the observations of our study.

This thesis work is intended to inform future documentation creation efforts. Insight into content correlation between different documentation types would be useful in determining what kind of information is required in documentation enrichment tasks that use information in external resources to improve documentation. Further, our proposal for information reuse automation would reduce the time spent by authors in the creation process, with the by-product of promoting consistency in content between different documentation types.

1.1 Contributions

The contributions of this work are:

1. The identification and characterization of the different match types of sentences in instructional documentation with respect to those in API reference documentation. A

sentence in instructional documentation is considered to have a match in the API reference documentation if there exists a sentence in the reference documentation which is semantically and/or syntactically similar. For example, an instructional documentation sentence may be *exact*ly matched to a reference documentation sentence if both texts are the same.

2. The elicitation of information reuse patterns observable in instructional documentation. A reuse pattern describes the construction and intention by which information from the API reference documentation has been used in instructional documentation. For example, *Method Description* is a pattern in which sentences from the method description in the reference documentation are embedded in the instructional documentation.
3. The proposal for automation of these reuse patterns to support instructional documentation creation. We propose parameters by which creation of reuse pattern instances can be automated while providing flexibility to the documentation writers.
4. An assessment of the impact of automation of the reuse patterns. We describe the extent of variance of the current pattern instances, had they been automatically generated.

1.2 Thesis Organization

The remaining thesis is structured as follows. Chapter 2 describes the software documentation environment, i.e. the structure of documentation as provided by Java and Python. We provide the context for the dataset and discuss its availability and characteristics. This chapter then discusses previous related research work. Chapter 3 presents an overview of our research method that includes dataset preparation and data analysis. Chapter 4 introduces the target documentation used in our case study as well as the procedure to collect and make ready the corpora for this work. Chapter 5 describes our data analysis procedure in detail. Chapter 6 presents the results and observations of our analysis. Chapter 7 presents the conclusion and discusses future directions of work.

2

Background and Related Work

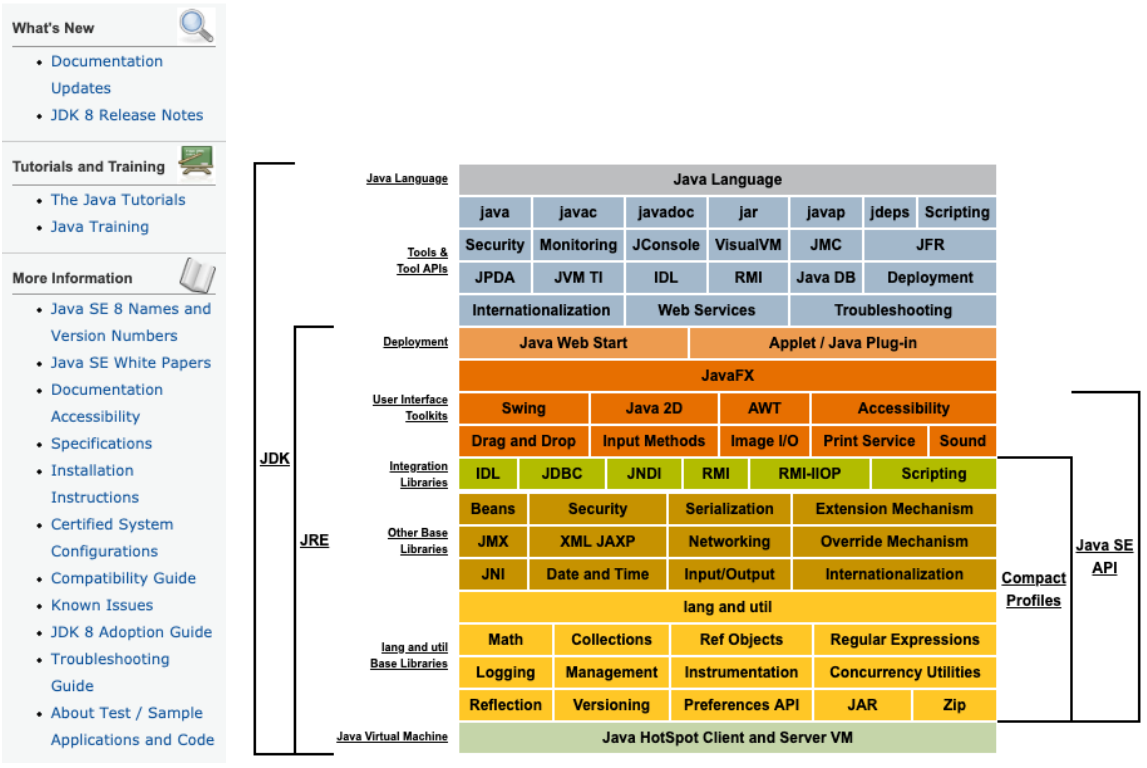
This chapter introduces the context for documentation related research (Section 2.1). It then describes previous research work that is related to the study done in this thesis (Section 2.2).

2.1 Background

Forward and Lethbride describe a software document as *any artifact whose purpose is to communicate information about the software system to which it belongs* [13]. This means that it is a source for users to learn and understand the capabilities and usage of the software to which it refers. However, the presentation of information in a software document varies based on its target user and intent. Further, documentation can also be categorized based on the association of the author to the software. Documentation written by the software developer or the owning organization is considered *official* documentation, whereas that written by enthusiasts and users are *unofficial*. As a result, a number of different types of documentation exist. In this work, we focus on the *official* instructional and API reference documentation of Java and Python.

Java is a programming language, released also as a development toolkit, which is well known to be extensively documented. Java's developer corporation, Oracle, provides API documentation, tutorials, training documentation as well as additional references like the Java Development Kit (JDK) Adoption Guide and installation instructions [1] (all footnotes are presented in References). Figure 2.1a displays the table of contents for the Java Platform

2.1 Background



(a) (b)

Figure 2.1: Java Documentation Index [1]: (a) Table of contents for the Java Platform Standard Edition 8 Documentation (b) Conceptual diagram provided by Java describing the different products provided by the Standard Edition. Each text in the picture is a link to the documentation of the product. Copyright © 1995, 2019, Oracle and/or its affiliates.

Standard Edition 8 Documentation, and is accompanied by a conceptual diagram shown in Figure 2.1b of the products provided, in which each text is a link to its corresponding documentation. As can be seen in the figure, Java provides a number of documentation types, from white papers to version notes, and from API reference documentation (on-click of Java SE API in Figure 2.1b) to tutorials and training material. Since these documents are provided by the developer corporation, they are *official* documentation.

Python documentation, as can be seen in Figure 2.2, provides a similar, but different set of documents [2]. *Library Reference* is analogous to the *Java SE API documentation*, and tutorials and installation manuals are provided in both cases. Unlike Java, however, Python does not provide documentation on known issues, but does contain a set of HowTos that

2.1 Background

Figure 2.2: Python Documentation Index

go in depth into certain topics. Python, being open source, receives the documentation available on the official webpage from developer contributors and hence can be considered *official* in nature.

Previous work by Watson [42], Garousi et al. [15] and Angelini [7] have all indicated the existence of different documentation types to support software and have studied their occurrences for software and their usage during development and maintenance tasks. However, these too, vary across the software being studied. While standards exist for documentation of computer software [30] [32], there is no set standard for structuring documentation sets, or collections of different documents, across languages and their APIs.

Application Programming Interfaces (APIs) are functions and procedures that perform specific tasks. These tasks can vary from creating and accessing services to functionality for building applications like *calendars*, to more complicated machine learning algorithm implementations that allow developers to, at the very least, simply feed in data and

perform learning tasks. Usually these methods are organized into classes, packages and modules based on the functionality they provide. An example of an API in Java is the `java.util.regex` [3] package which provides a set of classes to match text against regular expression patterns.

API reference documentation describes the structure and behaviour of API elements such as classes, methods, fields etc. along with information about input and output of these services that a user must be aware of when applying the API. In general, API documentation tends to follow the structure of the API itself - first describing overall components and then sub-components within them.

We use the term **instructional documentation**, consistently with Fourney and Terry [14], to describe documents that present how functionality provided by the language can be used, usually using specific applications as examples. Code fragments which perform certain tasks accompany explanations, to demonstrate how to use different components such as APIs or language syntax for different scenarios. The structure of such tutorials is generally narrative and interactive in nature and structured according to the preferences of the documentation writer. Instructional documentation in our study is comprised of tutorials in Java and tutorials and How-Tos in Python. While Java tutorials about a single API span multiple pages and are separated as individual sections, Python instructional documentation are structured in a single web-page. This type of documentation can be seen as analogous to user manuals from the work by Garousi et al. [15] and cook book-like documentation from Meng et al's work [26].

While API reference documentation has been the focus of most prior work (see Section 2.2), instructional documentation has been far less explored. Fourney and Terry, in 2014, focused on extraction of tutorial data for automatic processing and concluded that the research community needs a well-defined coding system to categorize information in tutorials [14]. To this day however, little work has been done in this area, likely because, as they describe, a number of language processing challenges arise in the investigation of such documentation.

2.2 Related Work

This section reviews previous research that is most related to that done in this work. Prior work has attempted to categorize the different types of documentation (Section 2.2.1), evaluate their quality (Section 2.2.2), assess the needs and wants of users (Section 2.2.3), improve their quality (Section 2.2.4) and study and propose patterns for documentation reuse (Section 2.2.5). Here, we also describe how this thesis augments and differs from related research work in the field of software documentation.

2.2.1 Types of Documentation

There exists little known prior work dedicated to categorizing and differentiating documentation types. Sommerville describes the different types of documentation that complement the software process [37]. He also makes suggestions for effective writing of these documents. His textbook distinguishes between five types of software product documentation based on the types of users to which they must cater and their relative experience levels, namely, *functional description*, *installation document*, *introductory manual*, *reference manual*, and *system administrators guide*. We focus on introductory manuals and reference manuals which Sommerville's work describes as targeting end users. Other prior work simply recognizes that different documentations exist. Garousi et al. state that there are two types of software documentation - technical documentation and user manuals [15]. Intuitively, technical documentation for languages would elicit facts of the programming language in question and could be in the form of embedded comments in code and API documentation. User manuals on the other hand, are tuned towards usage of the language data structures and libraries, in general or for specific use-cases. User guides, example documentation and tutorials may be considered as this type of documentation.

Developers often draw upon multiple different resources while learning a new language. Parnin and Treude studied the type of information sources of the top 10 web search results for JQuery API methods [29]. They identified 10 frequent kinds of resources: *code snippet site*, *Q&A*, *forum*, *official bug tracker*, *mailing list entry*, *official documentation*, *official forum*, *unofficial documentation*, *stackoverflow* [4], *blog post* and *official API*. While for 99.4% of methods tested, the official API documentation was in the top ten, other official docu-

mentation appeared only for 30.1% of methods. Blog posts were the next most frequently occurring at 87.9%. Deeper investigations discovered that among blog posts, about 49% of them were tutorials. Prior work has not provided a comparison between these documentation types. While this thesis focuses on *official* documentation, our analysis method can be extrapolated to *unofficial* ones as well.

2.2.2 Evaluation of Documentation

Watson developed a heuristic to evaluate whether API reference documentation contains important elements that help developers learn [42]. This work specifies its application to new and returning developers who have sufficient knowledge about how to code their task but only want to learn the features of the new API. At a broad level, Watson et al. evaluated documentation sets, i.e. collection of different types of documentation for a software, based on *initial impression*, *experience* provided to a reader and any *additional data* that exists [41]. One of the emergent observations of the study was that documentation components that developers prefer such as tutorials and sample applications were found in less than half of the 35 libraries studied [33, 34].

Angelini studied the API documentation of eight web applications with the intention of better understanding technical writing patterns [7]. While the work focused primarily on API documentation, one of the subsequent findings was that all the web applications studied contained at least one among an *Overview/Introduction*, a *Get Started*, a *Best practices/Usage guidelines* and a *Tutorial*, collectively referred to as *additional documentation*. However, no one web application comprised the full set of supplementary additional documentation. This confirms Watson's previous study of the presence of incomplete documentation sets.

Most recently, Aghajani et al. performed a large-scale empirical analysis of 878 discussions across 4 sources (mailing lists, Stack Overflow [4], issue repositories and pull requests) of users describing issues in documentation [2]. The outcome was the development of a taxonomy of documentation issues, classified into four main categories related to what information content is written, how it is written, issues related to the documentation process and documentation tool issues.

2.2.3 User Needs and Wants of Documentation

The study of developer needs has been a continuous one. Early work by Robillard discovered that despite the existence of multiple resources, developers need good instructional materials, with code examples and sample applications [33]. Robillard and Deline performed a combination of surveys and interviews to ultimately establish five integral factors to be considered during the API documentation creation process - *documentation of intent*, *code examples*, *matching APIs with scenarios*, *penetrability of the API*, and *format and presentation* [34]. Uddin and Robillard conducted an exploratory study to determine the shortcomings of API documentation and presented that 74% of documentation problems reported by 323 IBM software developers were caused by its content, including but not limited to incompleteness and ambiguity of API elements [40]. These pieces of work elicited that among other preferences, developers would like more explained examples, which are usually found in instructional documentation.

Garousi et al. further analyzed the usage and quality of “technical documentation” during the development and maintenance of products at a company [15]. Based on surveys of 25 participants, they concluded that, in industry, technical documentation (including but not limited to requirement specifications, design documents, source-code comments), source-code, communication with teammates and developers’ existing knowledge are all approximately equally used during the development process. This confirms developers’ multi-resource use, and calls for an analysis into the complementary nature between different types of documentation.

The thesis work performed by Josyula and Panamgipalli involved conducting interviews to determine a fixed set of information needs and information sources [31]. Upon which, an online survey was created to ask even more developers details about these identified needs and sources. They discovered that for designing product architecture, learning new programming skills and clarifying requirements, API reference documentation and online tutorials were frequently used information sources.

Meng et al. performed an extensive interview and questionnaire to determine developers needs, wants and experiences during the learning of a new API [25]. They discovered that most people when looking at a new API, ask “What can I do with this API" as their first

question. This is also analogous to specific scenarios that tutorials and user guides can support. Meng et al.'s work discovered that most people prefer using a getting started guide or working through code examples rather than looking at official documentation. Another finding from this work is that a majority of people perform Google searches instead of going straight to official documentation when they face an issue using an API. In general, it seems that developers learning a new API look through its reference documentation but would like more code and scenario-based examples. In addition, Meng et al. discovered that participants spent 49% of their development time looking at the documentation [26]. They also observed that API reference documentation and cook book-like documentation were used nearly equally frequently during the development process.

2.2.4 Assisting Developers and Improving Documentation

Rupakheti in his PhD thesis, addressed the resource switching of developers [36]. He explained that new developers find it difficult to formalize their requirements into an effective search query that can be used to browse online documentation resources. For this common, yet broad problem, he created a critic system called *CriticAL (A Critic for APIs and Libraries)* that provides recommendations and descriptions for client code using the API. *CriticAL* is based on API usage rules derived from patterns identified in common problems faced by developers while using the Swing framework.

To aide developers during the development process, Treude et al. developed an extraction mechanism to retrieve passages in documentation that describe how to perform a certain task using part-of-speech tagging and grammatical dependencies between words [39]. To realize the application of their work, they built TaskNavigator, an interface tool for users to refer to these compiled list of tasks. Two developers rated the extracted tasks, resulting in 70% of them being meaningful to at least one of the two developers. Hence previous work shows that scenario-based instruction is an integral resource for developers, in addition to reference documentation.

From early on, research has been motivated to improve documentation, including work done to augment information from other resources including source code [20] and *StackOverflow*, making inferences from documentation that are less explicit [44], highlighting

directives in documentation based on previously identified directives and user-input [11]. Treude and Robillard took advantage of content similarity between software artifacts to use supervised machine learning techniques to identify and recommend *insight sentences* from *StackOverflow* in documentation [38]. They used the value of the cosine similarity between a potentially informative *StackOverflow* sentence and sentences in the API documentation as a part of their feature set. Similarly, Jiang et al. built a model to identify fragments from tutorials that are relevant to the corresponding APIs [16].

The most in line with the goals of our work, is that of Oumaziz et al. who studied the reuse of documentation tags in source code used to generate reference documentation [28]. They created a duplication detector to identify the duplicate documentation tags in seven Java APIs that use JavaDoc and report that the most commonly duplicated tags are `param` and `throws` where 20% to 40% of these tags are duplicated. They performed a qualitative content analysis to determine if these duplications are intended or not and discovered that at least 57% were unintended "copy-pastes". They further proposed a simple documentation tag reuse mechanism to avoid duplicate information in documentation.

2.2.5 Patterns for Documentation

The earliest, most well-known work for patterns in documentation is that of Alexander who created a pattern language for common people to be able to design their own homes without the technical skills of an architect [6]. This work has become a foundation for patterns in fields beyond home construction. Johnson incorporated and modified Alexander's patterns to the domain of software engineering, specifically software frameworks [17]. Aguiar and David have also taken inspiration from Alexander's pattern language and performed in-depth research on patterns to document frameworks, releasing their work in multiple parts [3, 4, 5]. Other work on documentation patterns for software frameworks includes that done by Butler et al. which provides suggestions by which a framework's use and reuse in other applications can be documented [8, 9].

With respect to reuse of documentation, prior work has attempted to bridge the gap between software clone detection and software documentation to identify and extract duplicate textual information in documents. These are useful in indicating to documentation

creators the existence of redundancies or inconsistencies as well as in documenting other software elements which are similar to ones already documented. Luciv et al. and Koznov et al. created tools based on their proposed processes to automate the detection of repeated fragments of text in technical documentation [22, 18]. They also proposed methods by which the document can be modified and refactored based on the texts identified, to improve the quality of documentation. Koznov et al. also proposed a mechanism by which duplicates can be managed in documentation and efficiently taken advantage of, in documentation creation and maintenance [19]. Luciv et al. present an algorithm to detect "near-duplicates" in all documentation types from design specifications to user guides and discuss the strengths and weaknesses of the approach after a rigorous evaluation and manual analysis [23]. All these works perform duplicate detection *within* a single documentation type and propose changes and refactoring on individual documents. On the other hand, we analyzed the reuse of textual information *across* two documentation types and in our work, propose patterns to reuse this information to aid documentation generation and promote consistency across documentation of APIs.

Previous related work has mainly focused on content and structure of API reference documentation. Fourney and Terry described the challenges presented when attempting to dissect instructional material for automated understanding and processing [14]. They found the need to develop a formalization of the content present in a tutorial with the purpose of templating online tutorials. While a number of work has focused on doing this in API documentation [24] [27], tutorials seem to be far less explored, possibly because, as Fourney and Terry point out, even something as seemingly simple as determining what a *step* in the tutorial is, is a difficult problem.

In this thesis, we analyzed the information in instructional documentation with respect to the API reference documentation. This would help better understand programming language documentation practices and the trend of commonly occurring API documentation and lack of sufficient instructional documentation, despite developers having voiced their needs for such materials.

3

Overview of Research Method

We conducted a case study on similar API libraries in Java and Python. We performed a qualitative content analysis [21], to study the existing documentation, its characteristics and elicit emergent observations. Specifically, the complete procedure includes the following steps (as in Figure 3.1):

1. We first determined the API library topics for our case study (see Section 4.1).
2. We retrieved the documentations of these topics and performed a number of preprocessing steps in order to make the text usable for our study (see Section 4.2).
3. We then extracted individual sentences from each of these instructional documentations (see Section 4.3).
4. We manually analyzed each of the sentences extracted to determine if they contain information *related to APIs* or not, and classified them according to the relative relatedness to sentences in reference documentation (see Section 5.1).
5. We characterized these matches based on their relative position in the documentation (See Section 5.2).
6. Based on our observations of information reuse from API reference to instructional documentation, we identified some common information reuse patterns and discuss here their instances in the documentation studied (see Section 5.3).
7. We characterized instances of these patterns in each documentation based on the strictness with which they adhere to the patterns associated. With the intention of

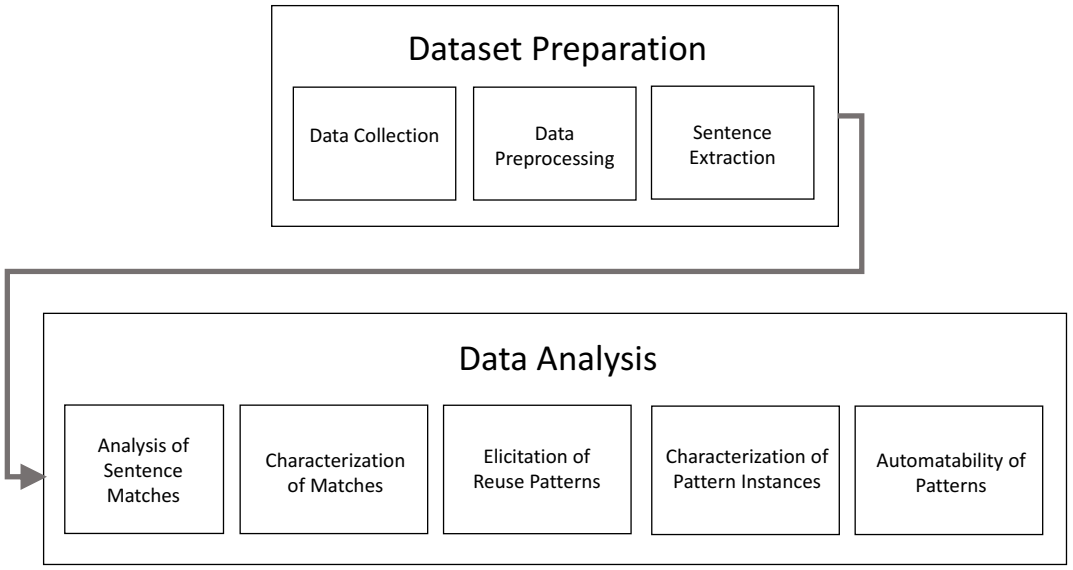


Figure 3.1: Case Study Approach

mitigating documentation creation efforts, we formalized transformations in order to accommodate these patterns automatically (see Section 5.4).

8. We assessed the impact of automation of these patterns on the existing documentation (see Section 5.5).

We chose to analyze documentation in Java and Python because of the vast difference in both programming languages. Both were initially developed with different programming paradigm support. While Java began as an object oriented programming language, and has now grown to accommodate the functional paradigm, Python supports object-orientation, functional, imperative and procedural programming styles. The difference in syntax between the two languages provides for insights on whether documentation patterns can be seen across languages with different APIs and code syntax. Further, Java developers are required to adhere to a certain format to make their code documentation-ready. The *javadoc* tool that is bundled with the Java development kit is responsible for generating API reference documentation based on this format. As a result, Java reference documentation is standardized in its structure. Tutorial-like instructional documentation does not follow any explicit documentation structure or standard. Python, on the other hand, does not enforce

documentation structure standards, though they do specify the style that should be followed [47].

The only known prior work of direct comparison between documentation in Python and Java is that of Wildermann [43] who reproduced and expanded on the work by Maalej and Robillard [24] which identified knowledge types in Java API documentation. Our work on applicability of documentation reuse patterns observed in instructional documentation based on corresponding reference documentation augments to previous cross programming language observations, providing insight into the extent of generalizability of relationships between documentation types in terms of their information content.

4

Dataset Preparation

We describe the dataset for our study and the data collection procedure (Section 4.1), steps taken to preprocess the data (Section 4.2) and the means by which individual sentences were extracted for analysis (Section 4.3) below.

4.1 Data Collection

In this thesis, we focus on textual *official* documentation of APIs in the Java Platform Standard Edition version 8 and Python version 3.7.2. We focus on three commonly used libraries: Regular Expressions (henceforth `REGEX`), URL connectivity and content processing (henceforth `URL`) and Input/Output (henceforth `I/O`) primarily because of the existence of the library APIs and their documentations in both languages under study. These libraries are based on deep-rooted concepts, such as regular expressions in the case of `REGEX`, as the foundation and motivation for their implementation. As a result, the documentation of these APIs is expected to not only address the API but also the foundational concepts behind them.

We studied instructional and API reference documentation (see Section 2.1) for these API topics, all of which are available as HTML files from the official websites of Java and Python. It is important to note that the actual implementations of APIs and the functionality they provide are likely to differ in Java and Python, even though they may be on the same topic. For example, the `URL` packages in Java do not directly correspond to those in Python. Connection via sockets is provided within the `java.net` package [5]. Whereas, in Python,

`socket` [6] is a separate module, outside the `urllib` package. For this work, we focus on the information reuse from API reference to instructional documentation and do not dwell upon the details of implementation of the APIs.

The details of all instructional documentation files studied and their mapped API reference documentation are available in Table 4.1. We used data triangulation, i.e. multiple data sources, in terms of programming language and API topic to strengthen the validity of our work [35].

Language	Doc. Type	Concept	Package or Module
Java Platform SE v8	API	REGEX	*java.util.regex* [7]
		URL	*java.net* [8]
		I/O	*java.nio.file* [9]
	Instructional	REGEX	*Lesson: Regular Expressions* [10]
		URL	*Lesson: Working with URLs* [11]
		I/O	*File I/O (Featuring NIO.2)* [12]
Python v3.7.2	API	REGEX	*re* [13]
		URL	*urllib* [14]
		I/O	*Built-in Functions* [15]
	Instructional	REGEX	*Regular Expression HOWTO* [16]
		URL	*HOWTO Fetch Internet Resources Using The urllib Package* [17]
		I/O	*Input and Output* [18]

Table 4.1: Documentation Dataset

4.2 Data Preprocessing

We decided to perform our analysis at sentence level granularity because a sentence is a cohesive unit of information that can be reused. To enable the case study at the sentence level, it was necessary to divide the instructional documentation into individual sentences. However, this process was not trivial due to the varying HTML structure across Java and Python. Difficulties arose because sentences were sometimes punctuated incorrectly at the end of a sentence, usually in the case of list items, or contained ending-like format mid-sentence, such as a period followed by a space as in 'e.g. X'. The issues faced are described in greater detail in Appendix A.

These issues were overcome by performing a number of preprocessing steps; the exhaustive list can be found in Appendix B. In order to accommodate for the varied structure of documentation, we parsed the HTML for both Java and Python differently. We also performed a number of preprocessing steps including removing content inside HTML *table*, *script* and *style* tags, replacing code snippets with a single token and treating words like 'e.g.' and 'etc.' specially.

4.3 Sentence Extraction

After all the preprocessing steps, we split individual sentences using a period or an exclamation followed by a space ('. ' or '! ') automatically. Because of the heavy use of most other characters to indicate regular expression tokens, we avoided splitting sentences on other punctuation. For example the question mark (?) refers to 0 or 1 occurrence of the previous character in regular expressions and is described in the `REGEX` API and instructional documentation. Many examples and definitions would then be unnecessarily split. This was also the case when using popular sentence splitting from natural language processing libraries like `nltk` [44] and `SpaCy` [45], motivating us to perform the splitting ourselves. The documentation of `I/O` in `Java`, too, describes `glob` string matching syntax, which is very similar to regular expressions. As a result, we chose to follow our extraction mechanism for all the documentations to maintain consistency. The restraining of question marks as a sentence divider resulted in few texts comprising of a question followed by a statement. For these cases, we manually separated them into two individual sentences. Similarly, when extracted texts were found to have multiple individual statements, these were also manually separated into distinct sentences. Further, in three scenarios in `REGEX`, two sentences divided because of an unfavorable split were manually merged to form a single, coherent sentence. An example is the following sentence in `REGEX` in `Python` instructional documentation,

> You can make this fact explicit by using a non-capturing group: (?:...), where you can replace the . with any other regular expression. [16]

which was previously split on the '. ' occurring before the word 'with'. It is important to note that our goal was not to design an efficient sentence splitter, as this is out of the scope

	Java	Python
REGEX	403	401
URL	183	135
I/O	1200	123

Table 4.2: Number of sentences extracted from the documents

of our case study, but instead to split sentences coherently for further analysis. As a result, manual intervention is both acceptable and desirable to obtain maximum accuracy.

Occasionally, leading sentences under highly nested subheadings were found to have been prepended with the subheading during the automatic sentence extraction. Similarly, the first sentence of a "note" in the documentation would be prepended with the word "Note". In such cases, we simply ignored the subheading or leading word while performing our analysis on the sentence. For example, the first sentence under the subheading `Basic File Attributes` of the `REGEX` instructional documentation in Java was extracted as:

> **Basic File Attributes** As mentioned previously, to read the basic attributes of a file, you can use one of the Files.readAttributes methods, which reads all the basic attributes in one bulk operation. [19]

In this case, we simply ignored the leading **Basic File Attributes** during our analysis.

As a result of preprocessing and subsequent sentence extraction, we extracted 403 sentences in Java and 401 sentences in Python for `REGEX` instructional documentation, 183 sentences in Java and 135 sentences in Python from the `URL` instructional documentation and 1200 sentences in Java and 123 sentences in Python for `I/O`. Table 4.2 summarizes these values.

5

Data Analysis

We performed qualitative content analysis to assess instructional documentation sentences and map them to those that provide the same information in API reference documentation. Dagenais and Robillard studied the evolution of software documentation via user interviews and analysis of document revisions [10]. One of the results was that developers and contributors of open source software projects typically begin by creating one type of documentation, and as the project evolves, work on other documentation types. They described that some contributors write *getting started* documentation first, which is analogous to instructional documentation in our work, while others begin with reference documentation. For this thesis, we focus on the latter scenario in which the API reference documentation was written first, and hence treat it as the source while instructional documentation is the destination for information reuse. Specifically, we hypothesize that certain sentences from instructional documentation could be or have been reused from sentences in the API reference documentation.

We performed our case study across two dimensions - *programming language and their APIs*, i.e. Java and Python and *topic*, formalized by the three popular API libraries - `REGEX`, `URL` and `I/O`. We do this to incorporate diversity in our corpora in terms of writing style and information content. Further, as can be seen in Table 4.2, each of these documentations are of varying length.

We used a largely inductive approach, collecting observations based on our analysis. We identified four information reuse patterns for instructional documentation based on our

observations. We believe that our methodology can be extrapolated to documentations of other topics with similar as well as possible new, interesting observations. We intend to pursue this expansion to other programming languages and API topics in future work.

For clarity, henceforth, we use the format of [programming language]-[topic] to refer to both the reference and the instructional documentations. For example Java-REGEX refers to the documentation of regular expressions API in Java.

5.1 Analysis of Sentence Matches

We began our analysis by looking at each of the sentences in the instructional documentation and manually annotating whether the sentence is *related to API* or not. Here, *related to API* refers to any piece of information that describes syntax, functioning, behaviour, usage and/or quality of the API under consideration, but excludes information about specific examples, way-pointing and segways. For example, we annotated the following sentence as *related to API*:

> By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. [20]

This sentence from Java-REGEX describes the behaviour of the API. On the other hand, the following sentence in the same documentation describing what is expected of a user learning the API is *not related to API*:

> You must learn a specific syntax to create regular expressions - one that goes beyond the normal syntax of the Java programming language. [21]

Specific examples provided in the instructional documentation are also marked as *not related to API*. The reasoning is because an example is a very specific scenario of API use, and hence is not be expected to be in the reference documentation. In many cases, these examples accompany or refer to code snippets in their explanations. The following sentence from Python-I/O demonstrates this:

> So if f is a text file object opened for writing, we can do this: CODE. [18]

where CODE replaces a code fragment in the original documentation as a part of our preprocessing (see Appendix B).

To ensure robustness of the manual annotation and to mitigate the threat of bias, we performed an iterative process with multiple annotators when coding the sentences. This also ensures observer triangulation, wherein more than one observer performs the study to increase precision of the work [35]. First, three annotators (the thesis author and her supervisors) individually coded a subset of 97 sentences in `Java-REGEX` and 98 sentences in `Python-REGEX` as *related to API* or *not*. These sentences were selected as ones in the instructional documentation that match the syntactic patterns identified in the work done by [27], as an initial simple filtering mechanism. Of these 195 sentences, 180 were completed by all three annotators, decisively as one of the two classes or as *incomplete*, i.e. lacking sufficient context to be annotated accurately. The Fleiss agreement score of these annotations is 0.82 [12]. The remaining fifteen sentences were identified as ambiguous or confusing by at least one of the annotators.

Among the sentences marked as unrelated to API, we marked the sentences that describe an overview of *API content*. Such sentences provide vague or high-level information regarding what the API component comprises of. Hence, though they may seem as API-related, they in fact are not highly informative. One such example is the following sentence in `Python-REGEX`:

> Pattern objects have several methods and attributes. [16]

We also identified those sentences that describe what the documentation offers, or *document content* such as

> Much of this document is devoted to discussing various metacharacters and what they do. [16]

from the same documentation file.

A second round of annotation involved all three annotators together discussing a set of 32 ambiguous sentences or those with disagreements in order to collectively conclude the

correct annotation for such sentences. Additionally, we made the decision to incorporate context of the sentence to ensure that *incomplete* sentences could be classified as either *related to API* or *not*. After this, a single annotator proceeded to completely annotate the entire corpora of sentences from all three topics under study.

For those sentences that are marked as *related to API*, the annotator then identified equivalent information-providing sentences in API reference documentation. Based on the sentence found, this annotator also assessed the extent to which the sentences in the two documentation types were semantically similar. As a result, we identified six types of matches between sentences in instructional documentation and their corresponding API reference sentence sources:

Exact match

The sentence in the instructional documentation is exactly the same as a sentence in the API documentation. For example, in `Java-REGEX`:

Instructional documentation:	**API documentation:**
By default, matching does not take canonical equivalence into account. [20]	By default, matching does not take canonical equivalence into account. [22]

Manipulated match

The sentences in both documentation types seem to be exactly the same except for non-functional words such as *usually* or *but*. For example, in `Python-REGEX`:

Instructional documentation:	**API documentation:**
Usually ^ matches only at the beginning of the string, and $ matches only at the end of the string and immediately before the newline (if any) at the end of the string. [16]	*By default*, '^' matches only at the beginning of the string, and '$' only at the end of the string and immediately before the newline (if any) at the end of the string. [13]

As in the above example, the only difference in the two sentences are the words in italics. These words, *usually* in the instructional documentation sentence and *by default* in the API documentation sentence, though can be interpreted differently, are intended to mean the same. Hence, it seems as though the sentence from the reference has been copied and then slightly manipulated. Generally, a manipulated match occurs around other manipulated and exact matches, and only rarely occurs in solitude (see Section 6.1). However, this does not mean that *manipulated* matches can never occur around other match types.

Rephrased match

Here, the sentence presented in the instructional documentation is semantically equivalent with one in the API reference but is phrased in a different way, for example made from active to passive voice or vice versa. In `Python-URL`, the following sentences describe default behaviour of the `Request` method in `urllib.request`:

Instructional documentation:	**API documentation:**
If you do not pass the data argument, urllib uses a GET request. [17]	The default is 'GET' if data is None or 'POST' otherwise. [23]

Partial match

Such matches are ones in which only some of the information in the instructional documentation sentence is presented in the API reference. The `Java-REGEX` documentation contains the following sentences:

Instructional documentation:	**API documentation:**
Both methods [lookingAt() and matcher()] always start at the beginning of the input string. [24]	public boolean lookingAt() Attempts to match the input sequence, starting at the beginning of the region, against the pattern. [25]

Here, the reference documentation explicitly states that the `lookingAt()` method begins at the beginning of the input string, however does not describe anywhere that the `matcher()` method does too. Hence only some of the information provided in the instructional documentation sentence can be sourced from the API reference.

Inferred match

These occur when an instructional documentation sentence is implied from information provided in the reference documentation, provided the reader has some domain knowledge. This type of match can be subjective as it greatly depends on the experience and expertise of a reader. The following sentences are from `Java-I/O`:

Instructional documentation:	**API documentation:**
The varargs argument currently supports the LinkOption enum, NOFOLLOW_LINKS. [19]	If the option NOFOLLOW_LINKS is present then symbolic links are not followed. [26]

Here, the instructional documentation states that the `varargs` argument of the `Files.readAttributes` methods accept the constant `NOFOLLOW_LINKS`. The API reference documentation makes no explicit statement. However, the `readAttributes` method description describes its behaviour if this constant was to be passed as an argument. An experienced user would likely be able to infer that `NOFOLLOW_LINKS` is an acceptable input and should be passed in the `varargs` argument.

No match

In this case, no match for the instructional documentation sentence is found in the API documentation. The following example is from `Python-I/O`, and introduces the concept of *serializing*:

Instructional documentation:	**API documentation:**
The standard module called json can take Python data hierarchies, and convert them to string representations; this process is called serializing. [18]	-

There is no sentence in the API reference documentation that describes what serializing is and hence this sentence has *no match*.

5.1 Analysis of Sentence Matches

During this process of identifying matches, it was critical that the annotator take into consideration the context of the sentence being matched. For example, the following fragment highlights a sentence in the `Python-REGEX` instructional documentation [16].

> Groups indicated with `'('`, `')'` also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to `group()`, `start()`, `end()`, and `span()`. Groups are numbered starting with 0. Group 0 is always present; it's the whole RE, so match object methods all have group 0 as their default argument. Later we'll see how to express groups that don't capture the span of text that they match.

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

There is a sentence nearly syntactically the same in the API reference documentation [13].

> `\number`
> Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, `(.+) \1` matches `'the the'` or `'55 55'`, but not `'thethe'` (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of *number* is 0, or *number* is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value *number*. Inside the `'['` and `']'` of a character class, all numeric escapes are treated as characters.

While both sentences, when looked at individually seem like *manipulated* matches with an inconsistency in information between the two documentations, this is in fact not the case. From the surrounding sentences and the example, it is evident the instructional documentation sentence refers to the method `group()` in which the index of the results begin at zero. On the other hand, the API reference documentation sentence refers to groups within the regular expression syntax. And so, instead of identifying this as an inconsistency, we determined this case as *no match*.

To minimize the presence of *false negatives* and *false positives* for *exact* and *manipulated* matches, we performed syntactic similarity between sentences in instructional documentation and API documentation for one topic, namely `REGEX`. A false positive is a sentence that does not have a match in the API reference documentation but is annotated as having

a match. A false negative is a sentence that is matched to an API reference documentation but was annotated as no match. After our first round of complete annotation of `REGEX` documentation, we used Jaccard similarity with a low threshold of 0.5 and extracted all the sentence pairs of the type `<instructional documentation sentence, API reference documentation sentence>`, that had a similarity above this threshold [1]. We found that sentence pairs identified by this threshold but not matched during our analysis (two in `Python-REGEX`, one in `Java-REGEX`) were justifiably not among the *exact*, *manipulated* and *rephrased* matches. For example, the following sentence in `Python-REGEX` instructional documentation:

> To match a literal '$', use \$, or enclose it inside a character class, as in [$]. [16]

has a high Jaccard similarity of 0.86 with the following sentence in the API reference documentation:

> To match a literal '|', use \|, or enclose it inside a character class, as in [|]. [13]

However, both sentences talk about different characters: '$' and 'l' respectively, and hence can not be considered *exact*, *manipulated* or *rephrased* matches for our work. Sentence pairs that were manually identified as matched but were not identified by Jaccard Similarity were also justifiable, usually due to the need of manual intervention during sentence extraction or because the instructional documentation is matched to a sentence in the reference documentation of a different API. Such cases were two in number in Python and sixteen in Java. These results justify that the manual matching done in our case study is reliable.

A visualization of the annotation hierarchy of instructional documentation sentences can be seen in Figure 5.1.

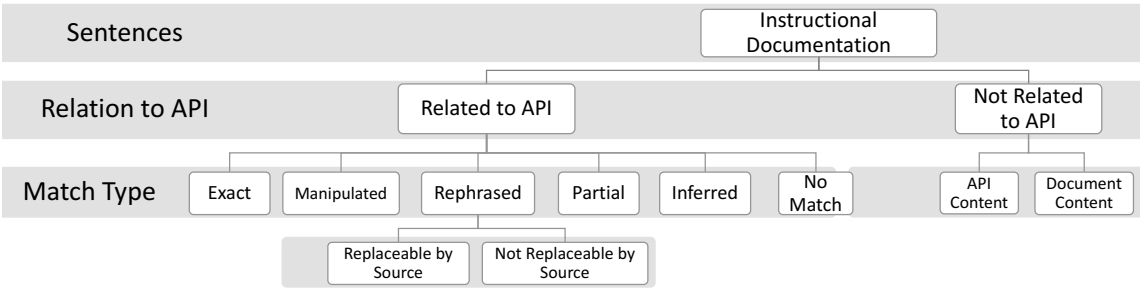


Figure 5.1: Hierarchy of Annotations of Sentences in Instructional Documentation

5.2 Characterization of Matches

We studied the instances of the different matches found and assessed the extent to which *rephrased* matches are necessary. We also performed an analysis on the sentences and their matches at the document level. We describe our analysis below.

We intended to understand the extent to which rephrasing sentences from API reference documentation in instructional documentation was necessary. To do so, we marked *rephrased* matched sentences in instructional documentation that are *replaceable* by their matched API documentation sentences without impacting the meaning of or removing information from the instructional documentation sentence (see Figure 5.1). For example in the `Java-URL` instructional documentation, the author writes

> A relative URL contains only enough information to reach the resource relative to (or in the context of) another URL. [27]

The original text in the reference documentation of which the above sentence is a rephrased match is:

> An application can also specify a “relative URL", which contains only enough information to reach the resource relative to another URL. [28]

Here, it seems as though this rephrasing was unnecessary as the source sentence provides

the same information as the sentence in the instructional documentation.

A *manipulated* match indicates that the difference in the instructional documentation sentence from the corresponding API reference sentence is minute and their semantics remain the same. Hence, all manipulated matched sentences in instructional documentation can be replaced by their matched counterparts without an impact on the meaning of the sentence.

As a result of this analysis, we categorized the match types that have a corresponding source into two major types:

- *Equivalent* - indicating the matched sentence provides the same information as its source. Such sentences in the instructional documentation can be replaced by their sources without altering the meaning of the text or disrupting its coherency. This includes *exact*, *manipulated* and *replaceable* matches.
- *Nonequivalent* - the matched sentence provides slightly less or more than its source or is written in a manner which would result in incoherency of the instructional documentation if replaced by its source. This includes *non-replaceable rephrased*, *partial* and *inferred* matches.

In the remainder of the text, we refer to these two categories, as well as *no match* and *not related to API* and specify lower level match types as required.

While our previous analysis focused on individual sentences, we also assessed the context and relative location of the sentence in the documentation. We studied the neighbouring sentences of each target sentence in order to better understand practices in instructional documentation information flow, and the relative position of occurrence of matches within each documentation. The emergent observations of match characteristics from this study are described in Section 6.2.

5.3 Elicitation of Reuse Patterns

The existence of *equivalent* matches implies information reuse from API reference documentation to the instructional documentation. We analyzed the instructional documentation

to identify possible patterns of this information reuse, with the intention of assisting documentation creation efforts by automating this reuse.

To determine the purpose, structure and variations of information reuse, we open-coded the "context" of all matched sentences in the respective instructional and API reference documentation, wherein there did not exist a pre-defined set of labels to choose from. For example the bullet points in the following text block from `Java-I/O` instructional documentation [29] are each considered as individual sentences and are each annotated with the context *catalog of class methods*.

The API consists of a few, easy to use, methods:

- `position` – Returns the channel's current position
- `position(long)` – Sets the channel's position
- `read(ByteBuffer)` – Reads bytes into the buffer from the channel
- `write(ByteBuffer)` – Writes bytes from the buffer to the channel
- `truncate(long)` – Truncates the file (or other entity) connected to the channel

The sentence at the first bullet point describing the method position has a *manipulated* match in the API reference documentation, which is shown below.

long position()
throws IOException
Returns this channel's position. [30]

This sentence from the reference documentation is annotated as *entire method description* because it forms the entire description of the method `position`. The contexts were formalized via an annotation session with the three annotators analyzing a random subset of sentences that are either *exact* or *manipulated* matches.

We then extrapolated these contexts to the remaining *exact*, *manipulated* and *rephrased* matched sentences in both Java and Python. We chose to omit *inferred* matches in this study of context because of the possibility of subjectivity and bias in inferring information as a result of the researcher's expertise in the domain. *Partial* matches were also omitted because we aim to assess information reuse information from API reference documentation and such matches contain information not found in the reference, as is the case with

no matches. We revisit *inferred*, *partial* and *no matches* while assessing the impact of automation of the patterns in Section 5.5. The list of unique *contexts* annotated can be found in Appendix C.

Based on the type of reuse and the contexts of matched sentences, we elicited information reuse patterns. We define a reuse pattern as a descriptive template for organizing sentences from API reference documentation in instructional documentation. These patterns are useful in supporting automation of documentation reuse and reducing the effort taken by authors to reproduce information presented in API reference documentation in instructional documentation. Each pattern is characterized by four properties:

- the **name** of the pattern,
- the **intent** for when the pattern is generally applied,
- the **structure** which indicates the template for organizing sentences from the API reference documentation in the instructional documentation,
- and a list of **parameters** to support automation of information reuse via this pattern. For example, for a pattern that extracts the leading sentence of a class, a parameter would be *class_list*, i.e. the list of classes for which this pattern is to be applied.

We then measured the number of **instances** of each of the reuse patterns, i.e. the number of occurrences of the pattern in each documentation. We further measured the total number of **sentences** in all instances of this pattern. We describe the patterns elicited as well as their occurrences in Section 6.3.

5.4 Characterization of Pattern Instances

To formalize the *extent* of reuse of a pattern, we characterize the instances of occurrence of a pattern in each instructional documentation with **pattern use type** as either *systematic* or *opportunistic*.

Systematic indicates a pattern that is followed consistently and it is clear which sentences in the structure of the API documentation are being chosen for reuse in the instructional documentation. In such instances, the pattern is usually at least semi-automatable and could support the generation of instructional documentation templates. Largely, *systematic* instances

are comprised of sentences that have *exact* and *manipulated* matches, though occurrences of other match types may occur in small quantities.

Opportunistic are ones where a reuse pattern could be abstractly present, but is cluttered with sentences not following the pattern and hence prone to be noisy in terms of reuse. These instances are comparatively more difficult to use for automated templating, because of the greater number of changes that would be made between current text and that post-automation.

If a documentation has multiple instances of a single pattern, some of which are *systematic* and some *opportunistic*, we default to the less-strict use: *opportunistic* to categorize the pattern use in the documentation. Hence, we define a pattern use type for each documentation studied.

5.5 Automatability of Patterns

Patterns in the reuse of documentation can support a degree of automation if these can be used to create documentation text from pre-defined templates. We studied to what extent this would have been possible in our six documentation cases.

The **parameters** property of each pattern can be used to support automation of usage of the reuse pattern in instructional documentation. As required by a documentation creator, the input values to these parameters can be varied in order to achieve the required output from the reuse pattern.

We assessed the **impact of automation** of implementing the reuse patterns based on these elicited parameters. That is, we simulated documentation reuse automation using the proposed patterns and analyzed the extent of difference between current reuse pattern occurrence and that which was produced in the simulation. Intuitively, *systematic* reuse instances would not be impacted drastically, whereas *opportunistic* instances could be significantly impacted. Further, *equivalent* matches would be maintained and the information in *non-equivalent* matches would be lost. To assess the possibility of a bloating factor upon automation, we also report the number of sentences that would be added from the API documentation into the automated pattern instance.

6

Results and Observations

In this chapter, we describe the instances of matched sentences identified in the documentation (Section 6.1) and their observable characteristics (Section 6.2). We then present the information reuse patterns observed, their instances (Section 6.3) and their emergent characteristics (Section 6.4). We also discuss the extent of automatability of these patterns in the documentations studied (Section 6.5).

6.1 Matched Sentences in Documentation Types

We found that in the Java instructional documentation of the three API topics under study, between 45% and 58% of the sentences are *related to API*. Python instructional documentation contains between 49% and 76% of such sentences. As shown in Figure 6.1, around half of the content in instructional documentation is expected to be present in the corresponding API reference documentation.

Figure 6.2 decomposes the occurrence of different match types with respect to all the sentences *related to API* in the respective instructional documentation of the three topics. In `Java-REGEX`, 37% of these sentences are *exact* matches of sentences from API documentation. Whereas in `Python-REGEX`, only 7% of the API-related sentences are *exact* matches. The number of *manipulated* matches in these documentations are relatively fewer, with 8% in Java and 5% in Python. The most frequently occurring match type in `Python-REGEX` is *rephrased* match, with 38% of sentences being characterized in this

6.1 Matched Sentences in Documentation Types

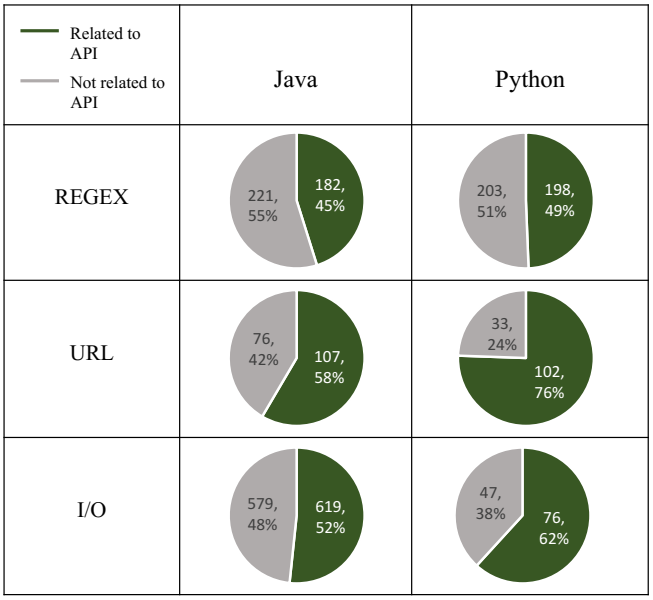
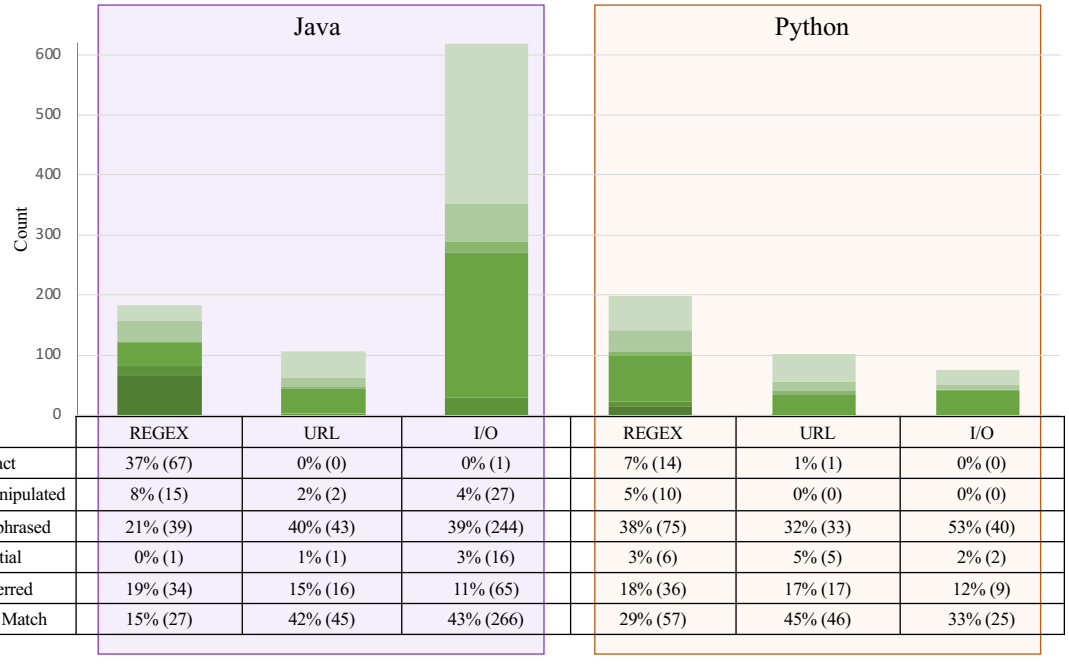


Figure 6.1: Distribution of sentences *related to API* in each of the documentations in the form of *absolute count*, *percentage*.



	Java			Python		
	REGEX	URL	I/O	REGEX	URL	I/O
Exact	37% (67)	0% (0)	0% (1)	7% (14)	1% (1)	0% (0)
Manipulated	8% (15)	2% (2)	4% (27)	5% (10)	0% (0)	0% (0)
Rephrased	21% (39)	40% (43)	39% (244)	38% (75)	32% (33)	53% (40)
Partial	0% (1)	1% (1)	3% (16)	3% (6)	5% (5)	2% (2)
Inferred	19% (34)	15% (16)	11% (65)	18% (36)	17% (17)	12% (9)
No Match	15% (27)	42% (45)	43% (266)	29% (57)	45% (46)	33% (25)

Figure 6.2: Comparison of match types between `REGEX`, `URL` and `I/O` in Java and Python.

way.

URL instructional documentation paints a different picture. No *exact* matches are found in Java, and only one was discovered in the Python documentation. Further, in Java, only two *manipulated* matches arose, while none can be found in Python. We found that 40% of sentences in Java and 32% of those in Python are *rephrased* matches. Surprisingly the majority of API-related sentences in URL in both programming languages have *no match* in the URL reference documentation with this case comprising 42% sentences in Java and 45% in Python.

The observations in I/O seem more comparable to URL than REGEX. In the documentation of this API in Java, only one *exact* match exists and 4% of API-related sentences have *manipulated* matches. Python, on the other hand has no *exact* or *manipulated* matches and has a majority of *rephrased* matches with 53% of sentences identified as this match type. In Java, *rephrased* matches are the second most frequent at 39% frequency. *No matches* tend to be the most frequently occurring with 43% such sentences, whereas in Python, *no matches* lie in-between that in REGEX and URL, at 33% frequency.

In all six documentations, partial matches are very infrequent (5% or lesser). It is difficult to conclude that a certain match type is the most prevalent or most frequently used, as this varies highly irregularly across the two dimensions of analysis.

6.2 Characteristics of Matches

Below, we present our results regarding the number of sentences that are unnecessarily rephrased, and a characterization of these rephrasings. We also discuss the cases in which a *rephrased* sentence cannot be replaced by its source sentence in the reference documentation. We then present other emerging characteristics of the matches that we observed during our inductive analysis.

6.2.1 Replaceable and Non-replaceable Rephrased Sentences

In REGEX, 51% of *rephrased* matches in Java and 43% in Python were found to be replaceable by their original counterparts. This scenario occurs in 47% of Java-URL sentences,

and 30% of `Python-URL` sentences. `I/O` shows similar numbers with 43% of *rephrased* sentences in Java and 30% in Python being replaceable. Figure 6.3 visualizes these distributions.

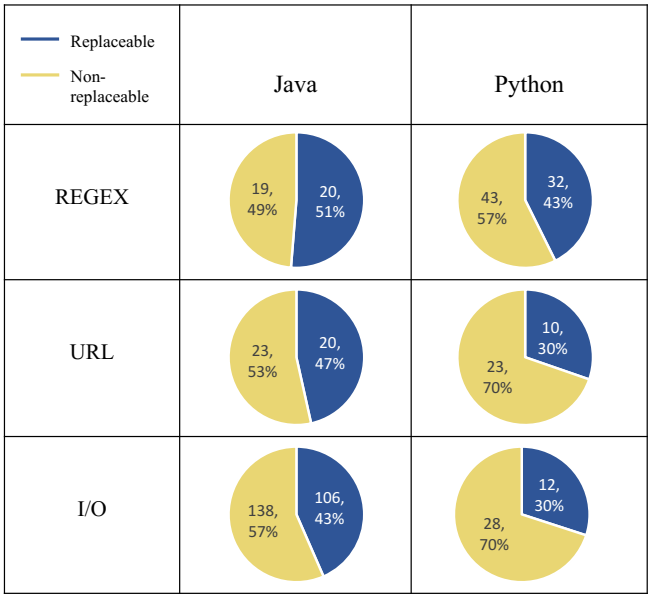


Figure 6.3: Distribution of *rephrased* matched sentences in instructional documentation that are replaceable by their matched counterparts in API reference documentation in the form of *absolute count*, *percentage*.

The large number of *replaceable* sentences could be attributed to the freedom of creativity of instructional documentation writers, giving them the opportunity to rephrase sentences if and when they wish, as no set standard for instructional documentation exists. This observation motivates automation of reuse patterns, as this would help maintain consistency across the documentation types and reduce the time spent by authors in unnecessary sentence modifications.

In all documentations, the percentage of sentences that cannot be replaced by their matched sentences is greater than half. The reason for the *non-replaceable rephrased* matches could be one of the following:

- The sentence is a rephrased version of two non-neighbouring API reference documentation sentences. As a result, these sentences cannot necessarily be systemati-

cally identified and merged without advanced mechanisms to merge the sentences coherently, efficiently and favorably for the reader in a human-like writing style. For example, the highlighted sentence in the following snippet from `Java-I/O` instructional documentation [31].

Both `newByteChannel` methods enable you to specify a list of `OpenOption` options. The same open options used by the `newOutputStream` methods are supported, in addition to one more option: `READ` is required because the `SeekableByteChannel` supports both reading and writing.

Specifying `READ` opens the channel for reading. Specifying `WRITE` or `APPEND` opens the channel for writing. If none of these options is specified, the channel is opened for reading.

The description in the reference documentation of `newByteChannel` method to which it refers mentions this information in two separate non-consecutive sentences, as highlighted in the screenshot below [26].

The `options` parameter determines how the file is opened. The `READ` and `WRITE` options determine if the file should be opened for reading and/or writing. If neither option (or the `APPEND` option) is present then the file is opened for reading. By default reading or writing commence at the beginning of the file.

In the addition to `READ` and `WRITE`, the following options may be present:

Option	Description
`APPEND`	If this option is present then the file is opened for writing and each invocation of the channel's `write` method first advances the position to the end of the file and then writes the requested data. Whether the advancement of the position and the writing of the data are done in a single atomic operation is system-dependent and therefore unspecified. This option may not be used in conjunction with the `READ` or `TRUNCATE_EXISTING` options.

Combining these sentences to generate a coherent sentence as in the instructional documentation is beyond the scope of our work.

- The sentence references or is in conjunction with a specific example. We would like to point out here, that sentences that provide example-specific information are marked as *not related to API*. However, sentences that provide general information about the API in the context of an example are considered as *rephrased* matches. `Python-I/O` instructional documentation [18] contains one such instance:

If you have an object `x`, you can view its JSON string representation with a simple line of code:

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Our preprocessing steps result in the sentence extracted in the following format:

> If you have an object x, you can view its JSON string representation with a simple line of code: CODE.

The code snippet within this sentence as seen from the screenshot informs that the `JSON` string representation of an object *x* can be viewed using the `dumps` method. The description for the method in the API reference documentation [32] states:

> `json.`**`dumps`**(*obj, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw*)
> Serialize *obj* to a JSON formatted `str` using this conversion table.

Hence, replacing this instructional documentation sentence by its match will result in a loss of the example.

- It introduces a use-case for the reference API documentation. For example, in `Java-URL` instructional documentation, the following sentence exists:

  > **After you've successfully created a URL, you can call** the URL's openStream() method to get a stream from which you can read the contents of the URL. [33]

  Here, the bold part describes when the `openStream` method can and should be used as opposed to the corresponding reference documentation that simply says:

  > Opens a connection to this URL and returns an InputStream for reading from that connection. [28]

  describing what the method performs.

- The matched API sentence may be providing excessive technical information. For example, the Java-REGEX instructional documentation states

  > The regular expression syntax in the java.util.regex API is most similar to that found in Perl. [21]

  On the other hand, the API reference documentation goes into deeper details:

  > The Pattern engine performs traditional NFA-based matching with ordered alternation as occurs in Perl 5. [22]

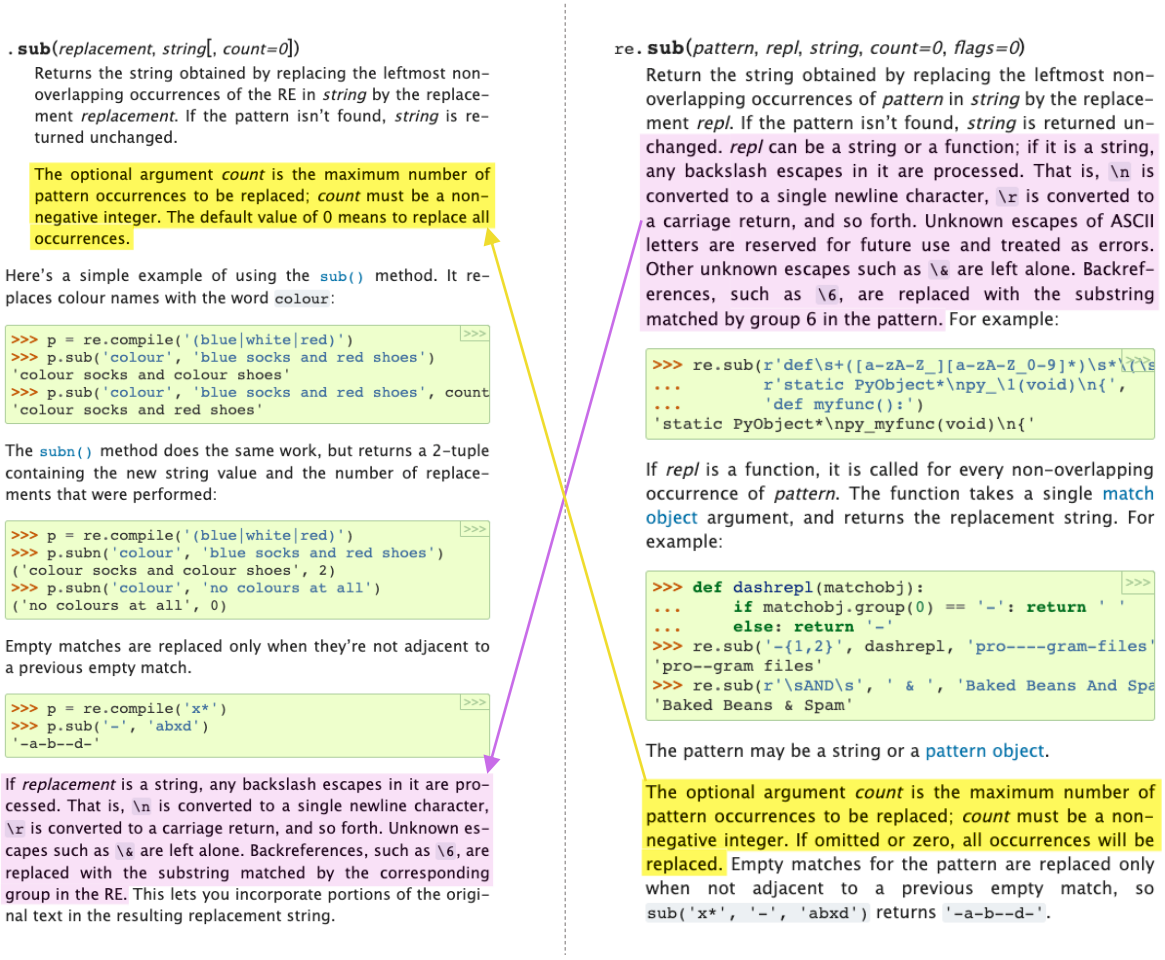
  In this case, the tutorial author might decide to rephrase this sentence to omit technical details from which a reader referring to instructional documentation would not be expected to benefit.

6.2.2 Positional Distribution of Different Match Types

In general, we found that the existence of the match types follow no specific pattern of occurrence in the chronology of the instructional documentation itself. *Equivalent* matches tend to be adjacent to one another, especially in Java-REGEX. Intuitively, this may be thought of as portions of inter-related text such as an entire method description that is copied from the reference documentation to the instructional documentation. This is more difficult to observe in the other cases because of the overall fewer *exact* and *manipulated* matches.

We also observed that the practice in Python-REGEX involves reordering, manipulating and rephrasing blocks of texts copied, in addition to expansion of these texts with examples. As a result, a mix of *exact*, *manipulated* and *replaceable* matches where seemingly the original source text could have simply been used, as is, and then augmented with examples is a commonly observed phenomena. The following screenshot shows the embedded description of the method `sub`. The left column contains text about the `Pattern.sub` method in instructional documentation [16] and the right contains a snippet from the API reference documentation [13] describing `re.sub`.

6.2 Characteristics of Matches



. **sub**(*replacement*, *string*[, *count=0*])

Returns the string obtained by replacing the leftmost non-overlapping occurrences of the RE in *string* by the replacement *replacement*. If the pattern isn't found, *string* is returned unchanged.

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. The default value of 0 means to replace all occurrences.

Here's a simple example of using the `sub()` method. It replaces colour names with the word `colour`:

```
>>> p = re.compile('(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub('colour', 'blue socks and red shoes', count
'colour socks and red shoes'
```

The `subn()` method does the same work, but returns a 2-tuple containing the new string value and the number of replacements that were performed:

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn('colour', 'no colours at all')
('no colours at all', 0)
```

Empty matches are replaced only when they're not adjacent to a previous empty match.

```
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b--d-'
```

If *replacement* is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes such as `\&` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by the corresponding group in the RE. This lets you incorporate portions of the original text in the resulting replacement string.

re.sub(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn't found, *string* is returned unchanged. *repl* can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes of ASCII letters are reserved for future use and treated as errors. Other unknown escapes such as `\&` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by group 6 in the pattern. For example:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s
...         r'static PyObject*\npy_\1(void)\n{',
...         'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

If *repl* is a function, it is called for every non-overlapping occurrence of *pattern*. The function takes a single match object argument, and returns the replacement string. For example:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files'
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spa
'Baked Beans & Spam'
```

The pattern may be a string or a pattern object.

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. If omitted or zero, all occurrences will be replaced. Empty matches for the pattern are replaced only when not adjacent to a previous empty match, so `sub('x*', '-', 'abxd')` returns `'-a-b--d-'`.

These two methods are comparable as the method definition for `Pattern.sub` in the reference documentation states that it is:

> Identical to the sub() function, using the compiled pattern. [13]

where `sub()` is a hyperlink reference to the documentation of `re.sub`. We can observe that, firstly, two arguments of the method have been described in a different order than in the original reference. However, this reordering seems unnecessary, as the change in order of description of the arguments does not hold significance in the instructional documentation. Secondly, the reference documentation contains all paragraphs (with examples) indented under the method definition. However, the same information provided in the instructional documentation, has indented only the first and second paragraph, while

the rest align under the parent section (not shown here). `Python-REGEX` contains two such method description embeddings, both of which display this indentation inconsistency. Whether this is intended or unintended is difficult to assess. This observation is unique to `Python-REGEX`, however, as no other documentation contains such method descriptions explicitly embedded in the content.

Figure 6.4 visualizes the position of *equivalent* matches, *nonequivalent* matches, *no matches* and *not related to API* sentences in the instructional documentation of the three API topics in both Java and Python. As it can be seen, the positional distribution is largely scattered. Often sentences with *no match* follow *equivalent* matches. As described in Section 6.2.5, this is usually because *no match*ed sentences describe *underlying topic information*, *internal working*, *behaviour* or *usage* of the API. Since most *equivalent* matches are class, method or constant descriptions (see Section 6.3), the observation of this ordering of match types is well-founded.

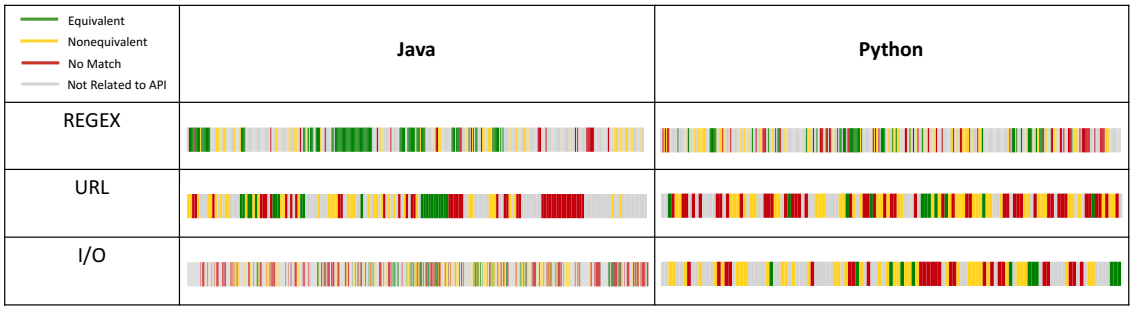


Figure 6.4: Position of match types in Java and Python in the instructional documentation of `REGEX`, `URL` and `I/O`

6.2.3 Redundancies

We discovered redundancies in instructional documentation, usually as a result of certain sentences being copied from the source and then slightly modified. This was then normally followed by another sentence that provides the same information, but likely written free-hand by the instructional document writer. For example, in `Java-I/O`, a section describing how to convert a `Path` object as required contains a modified excerpt from the

API reference documentation of the `toAbsolutePath` method:

> The toAbsolutePath method converts a path to an absolute path. If the passed-in path is already absolute, it returns the same Path object. [34]

However, what follows this method description is an example, after which the input, functionality and return of the method is reiterated:

> The toAbsolutePath method converts the user input and returns a Path that returns useful values when queried. [34]

This is also noticeable in the *Splitting Strings* subheading in `Python-REGEX`, where the behaviour of the argument `maxsplit`, when it is nonzero, is described twice, one sentence apart [16].

A justification for why such redundancies exist is difficult to conclude. If they are assumed to be intentional, it would likely be a reminder for the benefit of the reader. However, usually reminders in both Java and Python are specifically highlighted as notes. If these are accidental, then it would be useful to inform authors during creation of this redundancy. Regardless of the case, we intend for automation of information reuse to assist in informing authors of the redundancy, so they are made aware of its existence and can make corrections appropriately, to improve the instructional documentation.

6.2.4 Information Inconsistencies

Additionally, we discovered that in each of `Python-REGEX` and `Java-URL` instructional documentations, one sentence provided information inconsistent with that in API reference documentation or was incorrect due to lack of complete information.

The `Python-REGEX` documentation describes that inside a regular expression,

> ... '∧' outside a character class will simply match the '∧' character. [16]

This is not the case; outside a character class, if '∧' is at the beginning of a regular expression, then it would indicate start of a string, and in MULTILINE mode, would match immediately after each newline, and would not match the '∧' character.

`Java-URL` poses an interesting case. Consider the sentence in the instructional documentation [27]:

> This code snippet uses the URL constructor that lets you create a URL object from another URL object (the base) and a relative URL specification. The general form of this constructor is:
>
> `URL(URL baseURL, String relativeURL)`
>
> The first argument is a URL object that specifies the base of the new URL. The second argument is a `String` that specifies the rest of the resource name relative to the base. If `baseURL` is null, then this constructor treats `relativeURL` like an absolute URL specification. Conversely, if `relativeURL` is an absolute URL specification, then the constructor ignores `baseURL`.

The API documentation [28], instead states:

> ```
> public URL(URL context,
>            String spec)
>     throws MalformedURLException
> ```
>
> Creates a URL by parsing the given spec within a specified context. The new URL is created from the given context URL and the spec argument as described in RFC2396 "Uniform Resource Identifiers : Generic * Syntax" :
>
> `<scheme>://<authority><path>?<query>#<fragment>`
>
> The reference is parsed into the scheme, authority, path, query and fragment parts. If the path component is empty and the scheme, authority, and query components are undefined, then the new URL is a reference to the current document. Otherwise, the fragment and query parts present in the spec are used in the new URL.
>
> If the scheme component is defined in the given spec and does not match the scheme of the context, then the new URL is created as an absolute URL based on the spec alone. Otherwise the scheme component is inherited from the context URL.

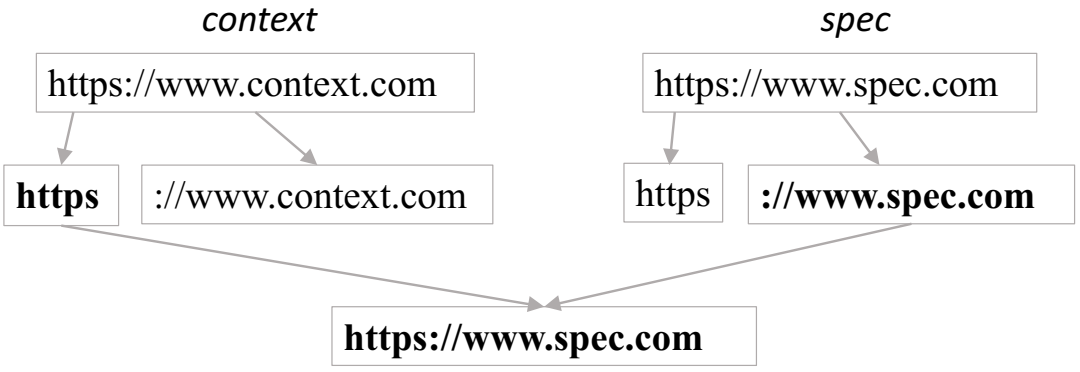
Here, `baseURL` is the same as `context` and `relativeURL` is the same as `spec`. In both cases, the resultant `URL` would be the same according to the highlighted sentences. However, both sets of sentences describe the internal working of the API differently. To illustrate an example, we use the terminology of the API reference documentation here for clarity. Consider a `context`:

https://www.context.com

and `spec`:

https://www.spec.com

Here, the `scheme` is `https`. According to the API reference documentation, when the `scheme` of the `spec` matches that of the `context`, then the `scheme` component is inherited from the `context`. That is, the API extracts the `scheme` component from `context`, removes the `spec`'s `scheme` and concatenates the former and latter as shown below:



Whereas, the instructional documentation states that if the `spec` is absolute (contains the `scheme` component) then the `context` is entirely ignored. So while the result in both scenarios would be the same because `scheme` is the same for both `context` and `spec`, the method by which the API performs this resolution differs in both descriptions. For that reason, we annotated this case as a *no match* for our work.

6.2.5 Reasoning for *No Matches*

As can be seen from Figure 6.2, the proportion of instructional documentation sentences that contain *no match* in reference documentation varies between 15% to 45% across the programming language and topic. The existence of a *no match* indicates a sentence that provides information about an API is newly introduced in instructional documentation and is

%	REGEX		URL		I/O	
Theme	Java	Python	Java	Python	Java	Python
Underlying Topic	**69**	23	29	22	19	12
Usage	15	12.5	**36**	**39**	**25**	24
Internal Working	0	**30**	2	4.5	16	0
Behaviour	0	2	11	26	9	**52**
Use-case	8	18	22	4.5	13	8
Performance	4	12.5	0	2	3	4
Version Info	4	2	0	2	4	0
Environment	0	0	0	0	7	0
API support	0	0	0	0	1	0
Input Configuration Details	0	0	0	0	3	0

Table 6.1: Distribution of information themes for sentences in instructional documentation having *no match* in percentage. For each document (i.e. each column), the most dominant theme is highlighted in bold.

lacking in the reference documentation. To better understand the reason behind the cases of *no match*es, we open coded the type of information that the sentences provide. We grouped the annotations into ten major themes: *underlying topic information*, *usage*, *internal working*, *behaviour*, *use-case*, *performance*, *version information and backward compatibility*, *environment and platform specific information*, *API support* and *input configuration details*. Table 6.1 summarizes the sentence distributions among these themes. We discuss each theme in detail below.

The majority of *no match* sentences in `Java-REGEX` provide information about the *underlying topic*, usually describing the general behaviour of the fundamental concept behind the API. The definition and description of a regular expression, its syntax, the behaviour of special characters or definitions of related terminology are examples of this theme that we found in instructional documentation but not in the API reference. For example, the following sentence defines a set of methods having similar functionality:

> Capturing groups are a way to treat multiple characters as a single unit. [46]

We discovered that *usage*, i.e. general information on how an API is expected or in-

tended to be used, is the most popular theme for *no matches* in `Java-URL`, `Python-URL`, and `Java-I/O`. The `Java-URL` instructional documentation recommends how to handle a `MalformedURLException`:

> Typically, you want to catch and handle this exception by embedding your URL constructor statements in a try/catch pair, like this: CODE. [27]

In `Python-REGEX`, the most commonly unmatched sentence theme is that of *internal working* with 30% of the sentences describing such information. For example, the following sentence was found in the instructional, but not reference, documentation:

> Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. [16]

A surprising 52% of sentences describe API *behaviour* in `Python-I/O`. It can be expected that descriptions of the way in which an API component performs is presented in the reference documentation, and so this finding is of interest. For example, the following sentence describes a particular behaviour of the `read` method on a file object:

> If the end of the file has been reached, f.read() will return an empty string (‘’). [18]

We found that sentences describing specific *use-cases* in which the API could be or is intended to be used, usually with the intention of motivating and justifying the usefulness of the API were also not matched with API reference documentation. This sentence from `Java-REGEX` is one such example:

> The split method is a great tool for gathering the text that lies on either side of the pattern that’s been matched. [20]

Sentences regarding *performance* of the API in terms of efficiency and scalability can also be observed in the instructional documentation, but not in the API reference documentation. The following sentence from `Python-I/O` is an example:

> This is memory efficient, fast, and leads to simple code: CODE. [18]

We observed that some sentences providing *version information and backward compatibility* were not matched in the API reference documentation. One example of a sentence providing information regarding content of a particular version is this sentence in `Java-REGEX`:

> As of the JDK 7 release, Regular Expression pattern matching has expanded functionality to support Unicode 6.0. [35]

This is a surprising finding because deprecation and enhancement information are generally specified in the API reference documentation, in order to caution developers about no longer supported API components, or introduce them to new ones. We expect that this kind of information can be found in the version release notes and we leave the exploration of this documentation type to future work.

Some of our observations are unique to `Java-I/O` documentation. This, we theorize, is likely due to its large length and diverse range of sub-topics, providing greater scope for writing style variation. We found non-matched sentences providing *environment and platform specific information*, *API support* and *input configuration details* information only in this documentation. While describing the typical syntax of a file location, the documentation provides the following platform specific information:

> In the Solaris OS, a Path uses the Solaris syntax (/home/joe/foo) and in Microsoft Windows, a Path uses the Windows syntax (C:\home\joe\foo). [36]

Another sentence describes whether a file system may be able to support the API components provided:

> A specific file system implementation might support only the basic file attribute view, or it may support several of these file attribute views. [19]

Sentences containing *input configuration details* information are ones which describe the structure of the input to an API. For example, in the `JAVA-IO` instructional documentation,

the `width` is an element of the format specifier in the `format` API. The sentence provides the following information about `width`:

> By default the value is left-padded with blanks. [48]

The default behaviour of this element of the format specifier is not mentioned in the API reference documentation.

We also identified one sentence describing a method for which the corresponding description in the API documentation was not descriptive enough to consider it as a match. While the instructional documentation states:

> visitFile - Invoked on the file being visited. [37]

The description of the `visitFile` method in the reference documentation is simply:

> Invoked for a file in a directory. [38]

While the sentences provide little explanation, the instructional documentation clarifies that this method is invoked *when a file is visited* as opposed to the reference documentation. Further, we found two more instances of descriptions in reference documentations that could have been matches for an instructional documentation sentence but were either incomplete or not clear in explanation. We consider both cases as *inferred* matches because their meanings can be deduced given familiarity with the API. One example is shown below:

Instructional documentation:	**API documentation:**
CONTINUE - Indicates that the file walking should continue. [37]	public static final FileVisitResult CONTINUE Continue. [49]

It is important to note that these themes are not exclusive for *no match* sentences. However, there are sentences which are matched to API reference documentation also providing information on these themes. We leave the detailed comparison of documentation on the theme level to future work.

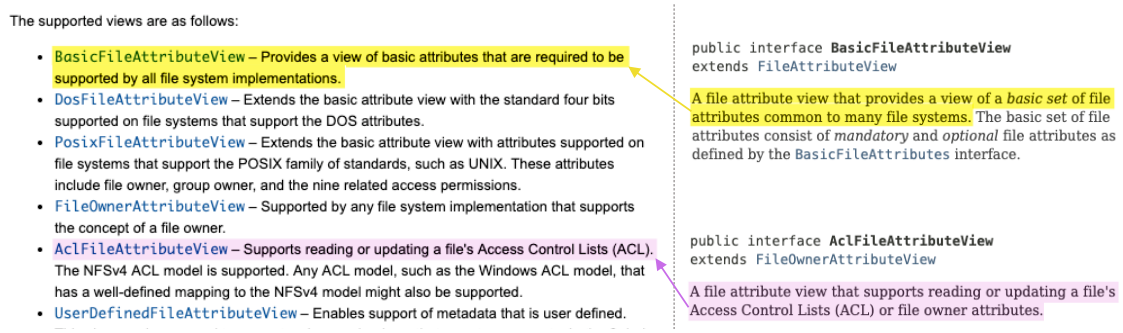
6.3 Elicitation of Reuse Patterns

During our analysis, we characterized the forms of reuse that occurred across multiple sentences, and sometimes multiple times. We call these recurring trends, documentation reuse patterns (as introduced in Section 5.3). For example, reused sentences in `Java-REGEX` often appear in lists of related methods in the instructional documentation to provide brief descriptions of those methods. We describe four elicited patterns based on our observations of information reuse from API documentation in the instructional documentation of `REGEX`, `URL` and `I/O`.

Class Description

Intent: Introduce classes in a package either as a sentence in a paragraph or as a list of classes in the package.
Structure: Extract the definition fragment, i.e. the leading sentence of the class description for each class and prepend the subject, which is the class name.
Parameters:

1. *class_list*: list of class names for which this structure is to be applied



The above screenshot shows an example of this pattern in `Java-I/O`. On the left is the instructional documentation [19] containing a list of interfaces of a particular functionality that are supported by this package. (An interface is a specific instance of a class.) The right column contains snippets from two of the interface descriptions [39] [40]. In this case, the

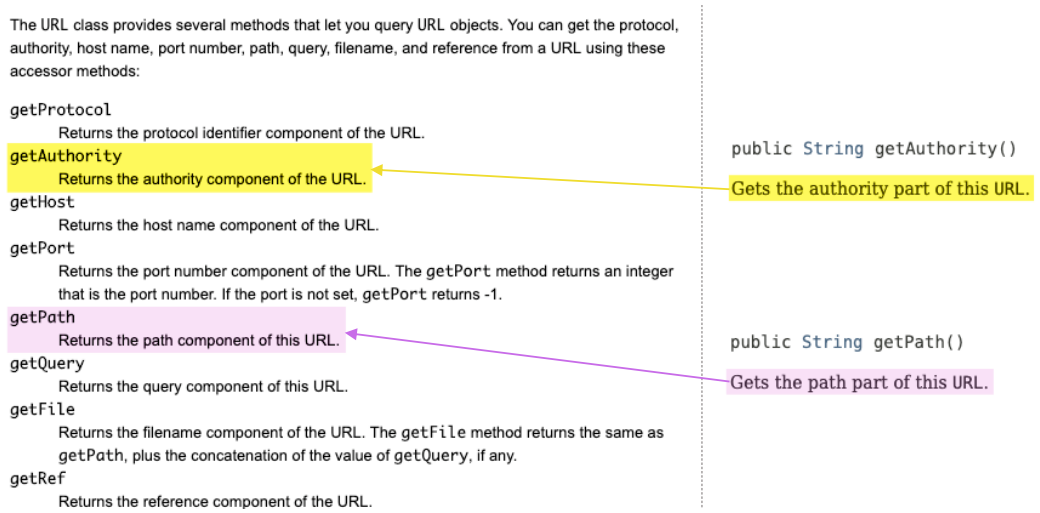
highlighted instructional documentation sentences match the corresponding sentences in the API reference documentation, which are the leading sentences of the interface description.

Method Description

Intent: To describe a collection of methods. Typical use of this pattern is to list methods in a class with similar functionality or based on the same topic, or simply to list all the relevant methods in a class, providing a short description for each method in the list. This is also used to list methods from a different API which provide similar functionality with the API being discussed primarily in the tutorial.
Structure: Extract method description of each method in the list.
Parameters:

1. *method_list*: list of methods to be included in this structure. Defaults to * (all methods in file).
2. *extractions*: sentences to be extracted from the method description in the form of (paragraph, sentence) where '*' defines 'all' and '[]' defines an inclusive range. For e.g.:
   - 1^{st} sentence in 1^{st} paragraph: (1,1)
   - 1^{st} sentence in every paragraph: (*,1)
   - 2^{nd} to 4^{th} and 6^{th} and 7^{th} sentences in 1^{st} paragraph: (1, [2-4,6,7])

   Defaults to (1,1).



The above screenshot is from `Java-URL` describing the list of accessor methods available for a `URL` object. The left column shows the instructional documentation [41], and the right has two snippets of the API reference documentation [28] with descriptions corresponding to the enlisted methods.

Constant Description

Intent: To provide a description for the constants passed as parameters to API methods or returned from a method call.
Structure: If within a method description, then extract the description of each constant in list from within the method description, else extract description from the generic constant description.
Parameters:

1. *constant_list*: list of constants to describe. Defaults to * (all constants in file).
2. *extractions*: set of sentences to be extracted from the constant description in the form of (paragraph, sentence) where '*' defines 'all' and '[]' defines an inclusive range. For e.g.:
    - 1^{st} sentence in 1^{st} paragraph: (1,1)

- 1^{st} sentence in every paragraph: (*,1)
- 2^{nd} to 4^{th} and 6^{th} and 7^{th} sentences in 1^{st} paragraph: (1, [2-4,6,7])

Defaults to (1,1).

The instructional document of `Java-I/O` [42] hosts an example in which the constants accepted by the `move` method are listed as follows:

> This method takes a varargs argument – the following `StandardCopyOption` enums are supported:
>
> - `REPLACE_EXISTING` – Performs the move even when the target file already exists. If the target is a symbolic link, the symbolic link is replaced but what it points to is not affected.
> - `ATOMIC_MOVE` – Performs the move as an atomic file operation. If the file system does not support an atomic move, an exception is thrown. With an `ATOMIC_MOVE` you can move a file into a directory and be guaranteed that any process watching the directory accesses a complete file.

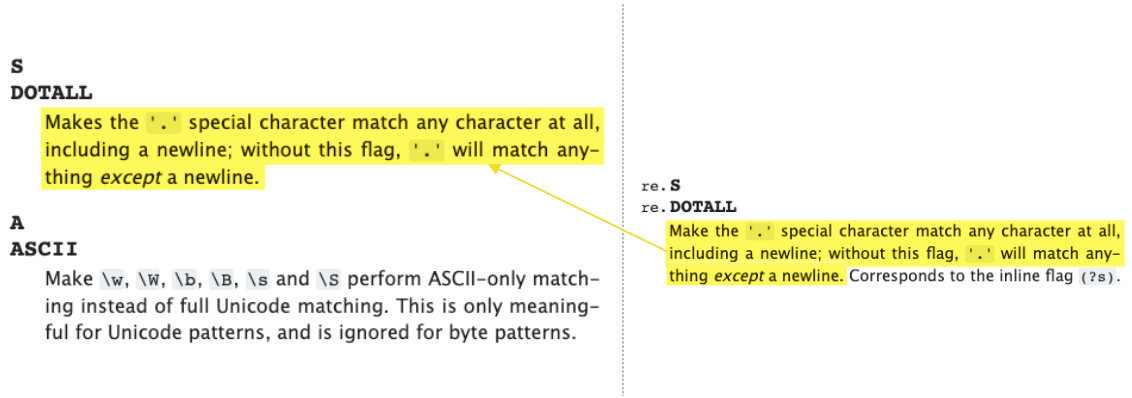
Though these constants can be used for multiple methods, and have their own descriptions in the class in which they are defined [43], they are listed in the `move` method description [26] with details describing their behaviour specific to the method. The screenshot of this list is shown below:

The `options` parameter may include any of the following:

Option	Description
`REPLACE_EXISTING`	If the target file exists, then the target file is replaced if it is not a non-empty directory. If the target file exists and is a symbolic link, then the symbolic link itself, not the target of the link, is replaced.
`ATOMIC_MOVE`	The move is performed as an atomic file system operation and all other options are ignored. If the target file exists then it is implementation specific if the existing file is replaced or this method fails by throwing an `IOException`. If the move cannot be performed as an atomic file system operation then `AtomicMoveNotSupportedException` is thrown. This can arise, for example, when the target location is on a different `FileStore` and would require that the file be copied, or target location is associated with a different provider to this object.

An instance in `Python-REGEX` instructional documentation contains a list of constants and their descriptions, which are sourced from a list of constants presented in the

reference documentation. The screenshot below has, on the left, a snippet from the list of constants presented in the HowTo [16], and on the right, the constant description in the library reference [13].



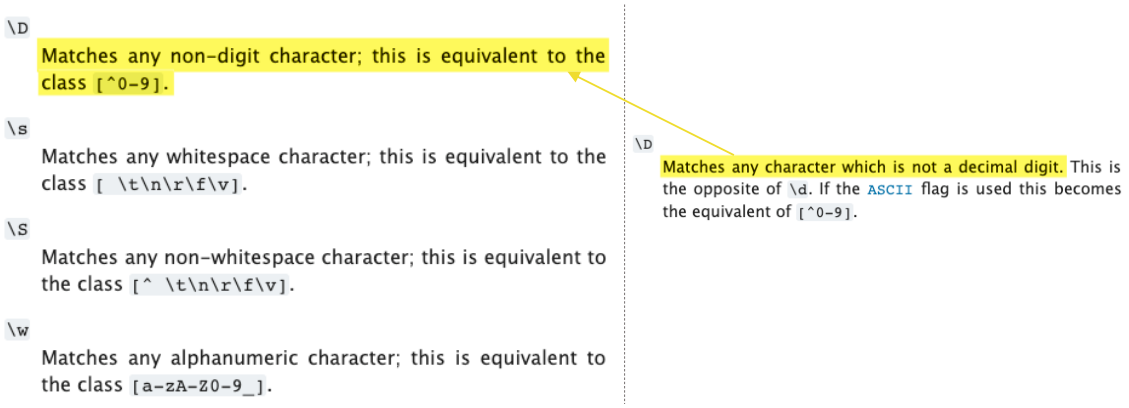
Input Configuration Options

Intent: To list and briefly describe the input configuration options. For example, structure of the regular expression language in `REGEX`, or structure of `URL`s.
Structure: Extract description of input configuration options from list in API reference documentation.
Parameters:

1. *input_list*: list of input configurations for which this structure is to be applied. Defaults to * (all input configurations in file).
2. *extractions*: set of sentences to be extracted from the constant description in the form of (paragraph, sentence) where '*' defines 'all' and '[]' defines an inclusive range. For e.g.:
   - 1^{st} sentence in 1^{st} paragraph: (1,1)
   - 1^{st} sentence in every paragraph: (*,1)
   - 2^{nd} to 4^{th} and 6^{th} and 7^{th} sentences in 1^{st} paragraph: (1, ([2-4],6,7))

   Defaults to (1,1).

The following figure contains, on the left, the list of input configuration options in Python-REGEX instructional documentation [16], of which the description of one option is highlighted. On the right, the source API reference documentation [13] from which the sentence can be reused has been highlighted.



6.3.1 Instances of Reuse Patterns

The instances of occurrence of the reuse patterns are summarized in Table 6.2. Java -REGEX and Java-I/O instructional documentation are the only two documentations which contain at least one instance of each pattern. Our characterizations of the pattern provide flexibility to accommodate instances which vary only slightly in terms of intention and content in the structure. To illustrate this tolerance, we take an example of variation in the instances of *Method Description* below, however this is applicable to all the four patterns elicited.

We noticed that five instances of the *Method Description* pattern in Java-I/O are not accompanied by descriptions. This phenomenon is common for other instances of *Method Description* in the same document as well as other documents. In fact, three of these instances describe the functionality of these methods in a sentence following the list. This is done especially in cases where the listed methods have similar functionality, differing only by the input parameters. The screenshot below shows one such instance [31].

> You can create a temporary file using one of the following `createTempFile` methods:
>
> - `createTempFile(Path, String, String, FileAttribute<?>)`
> - `createTempFile(String, String, FileAttribute<?>)`
>
> The first method allows the code to specify a directory for the temporary file and the second method creates a new file in the default temporary-file directory.

On the other hand, the instructional documentation of Python seems a lot less influenced by the original API reference documentation. `Python-URL` and `Python-I/O` show no instances of any of the patterns. `Python-REGEX` is an exception where there exist clear indications of reference API being explicitly embedded into the instructional documentation [16], like below:

> Another common task is to find all the matches for a pattern, and replace them with a different string. The `sub()` method takes a replacement value, which can be either a string or a function, and the string to be processed.
>
> **.sub**(*replacement*, *string*[, *count=0*])
>
> Returns the string obtained by replacing the leftmost non-overlapping occurrences of the RE in *string* by the replacement *replacement*. If the pattern isn't found, *string* is returned unchanged.
>
> The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. The default value of 0 means to replace all occurrences.

In the other two topics, references to methods are more often made in-line as follows:

> The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

Table 6.2: Instances of Information Reuse Patterns Observed

Pattern	Topic	Language	Cases Found	
			Instances	Sentences
Class Description	`REGEX`	**Java**	2	4
		Python	0	0
	`URL`	**Java**	0	0
		Python	0	0
	`I/O`	**Java**	1	11
		Python	0	0
Method Description	`REGEX`	**Java**	8	41
		Python	2	7
	`URL`	**Java**	1	11
		Python	0	0
	`I/O`	**Java**	10	35
		Python	0	0
Constant Description	`REGEX`	**Java**	1	35
		Python	1	24
	`URL`	**Java**	0	0
		Python	0	0
	`I/O`	**Java**	5	39
		Python	0	0
Input Configuration Options	`REGEX`	**Java**	2	6
		Python	3	42
	`URL`	**Java**	0	0
		Python	0	0
	`I/O`	**Java**	2	6
		Python	0	0

6.3.2 Discussion on Reuse Patterns

Some *equivalent* sentences in the instructional documentation are not covered by our proposed reuse patterns as listed in Table 6.3. In such cases, the individual sentences exist in sections regarding API sub-topics, usage or use-cases with examples. Such sections are generally narrative in style, and therefore difficult to break into fine-grained text structures (e.g. lists and tables). The matched sentences in the API reference documentation, too, are at arbitrary locations. This makes it difficult to formalize *intent* and a *structure*, and thereby generalize the sentence reuse as patterns.

% (#)	REGEX		URL		I/O	
Match Type	Java	Python	Java	Python	Java	Python
Exact	0% (0)	0% (0)	-	100% (1)	100% (1)	-
Manipulated	13% (2)	22% (2)	100% (2)	-	22% (6)	-
Replaceable	75% (15)	37% (12)	45% (9)	100% (10)	62% (66)	100% (12)

Table 6.3: Distribution of sentences in instructional documentation not belonging to any instances of any information reuse patterns. Each cell contains the the percentage with respect to the total sentences with that match type in that documentation and in parentheses, the absolute count. Hyphens (-) indicate there are no sentences at all for the match in the documentation.

We observed similarity to the pattern *Input Configuration Options* in `Java-URL` instructional documentation, however the structure is slightly different than other instances of this pattern. The API documentation of `REGEX`, in both Java and Python, summarizes input configuration options (comprising of `REGEX` language syntax), but the `URL` API documentation does not describe web `URL` syntax. Instead, for this instance, the configuration information is present within the class description. Similarly, a list of input configuration options describing the glob syntax in `Java-IO` can be sourced from the method description of `getPathMatcher`. However this source is simply a bullet point list and is difficult to extract particular configuration options from. Therefore, applying the *Input Configuration Options* in these scenarios would be unsuccessful because of the lack of a structured source.

Similarly, `Python-URL` contains another potential pattern instance. We observed fifteen sentences in instructional documentation *related to API* contained within a list of

classes in a module and their descriptions. Though it may seem that this could be an instance of the *Class Description* pattern, we discovered that most of these instructional documentation sentences describe the behaviour or usage of the class and do not have matches in the API documentation. Sentences that have matches are mainly regarding methods or attributes of the class and are rephrased. As a result, proposing a reuse pattern is difficult for these set of sentences. Further, the sections describing the purpose and functionality of API components (for e.g., *Openers and Handlers* [17]) are riddled with *no matches* as well as *rephrased* and *inferred* matches based on leading sentences of class and method descriptions. Given that these matches are largely unsystematic and inconsistent, developing a reuse pattern would be challenging in this case.

6.4 Characterization of Pattern Instances

We analysed the extent to which each instance of each pattern could be automated using the pattern parameters. We found that instances in `Java-REGEX` are *systematic* in nature, whereas in `Python-REGEX`, they are all *opportunistic*. `URL` and `I/O`, contain *opportunistic* instances, irrespective of the programming language.

An interesting observation regarding the pattern instances in the instructional documentation is that the intentions vary greatly even within a documentation. For example, `Java-REGEX` contains eight instances of the pattern *Method Description*, but the intentions vary including a catalog of methods in a class, catalog of methods with comparable behavior to those in a different class and catalog of methods with the same input type, to name a few. As a result, the decision of which methods compose the catalog require human involvement. The one instance of *Constant Description* in this documentation introduces the catalog of constants accepted by a method with the following two sentences:

> The Pattern class defines an alternate compile method that accepts a set of flags affecting the way the pattern is matched. The flags parameter is a bit mask that may include any of the following public static fields: [20]

However, upon inspection, we discovered that this list is missing the

UNICODE_CHARACTER_CLASS field which was added in Java 7. All the other fields, were available before this version as well. We can not assume that this exclusion is intentional or unintentional.

To assist authors in the instructional documentation creation process, we propose using the provided parameters to support the automated implementation of the reuse patterns. Having the parameters as input to our patterns gives the authors the freedom to decide which class, method, constant or input configurations to extract. We expect to build a tool that allows writers to choose a pattern and simply fill in the desired values for the parameters to ultimately generate a desired template for the instructional documentation as a base for the document. This would greatly reduce the amount of time and effort spent by authors to discover and retrieve the information they need from API reference documentation, and then copy and paste this to the instructional documentation. In addition, such a tool would ensure consistency between the information provided in both documentation types during their creation. Further, this would promote the development of standards for documentation reuse across software programming languages and different API topics, aiding readability for users as they learn new languages.

6.5 Automatability of Patterns

Table 6.4 describes the *impact upon automation* of information reuse using the patterns on the current pattern instances. Generally, *equivalent* matches are retained and *non-equivalent*, *no matches* and sentences *not related to API* are lost upon automation. However, two exceptions to this case arise. One method description instance in Java-I/O contains a *no match* sentence because the API reference description is inadequate (see Section 6.2.5). Upon automation, it would be replaced by this inadequate sentence. Similarly, in the same documentation, an instance of *Constant Description* contains the following sentence annotated as *not related to API* because it describes a specific example:

> The expression “a\u030A”, for example, will match the string “\u00E5” when this flag is specified. [20]

The description of the constant referred to, CANON_EQ, in the API reference documenta-

tion contains this exact sentence. Hence, upon automation, this sentence will be reproduced from its source in the reference documentation.

In general, the documentations containing *systematic* use of the patterns are least effected by the automation. For instances of all patterns except *Constant Description* in `Java-REGEX`, there would be no loss of information. For the instances of *Constant Description*, only one sentence *not related to API* would be lost.

Opportunistic instances, however, are prone to greater modification. For example, automation of the instance of the *Constant Description* pattern in `Python-REGEX`, would result in the loss of ten sentences. The instances of *Input Configuration Options* in `Python-REGEX` instructional documentation will be the most impacted, with 28 sentences being lost in total.

Our proposed automation does not mean to replace the role of a human documentation writer, but to provide them the organized content as a starting point in instructional documentation creation. The "lost" sentences can be incorporated back into the instructional documentation by the writers after the automatic application of the patterns.

Bloating, i.e., the increase of instructional documentation size, is a potential concern if a proportionally large number of sentences are added to the documentation upon automation. However apart from the instances of *Method Description* in `Java-REGEX`, the proportion of sentences added to sentences lost is lesser than one. This means that for most pattern instances, the number of sentences will not increase drastically upon automation.

Table 6.4: Impact on Automation of Information Reuse Patterns for Existing Instances.

Pattern	Topic	Language	Use Type	Retained	Lost			Added
				Equivalent Sentences	Non-equivalent Sentences	No Match	Not API related	# Sentences
Class Description	**REGEX**	**Java**	Systematic	4	0	0	0	0
		Python	-	-	-	-	-	-
	URL	**Java**	-	-	-	-	-	-
		Python	-	-	-	-	-	-
	I/O	**Java**	Opportunistic	9	1	0	1	0
		Python	-	-	-	-	-	-
Method Description	**REGEX**	**Java**	Systematic	41	0	0	0	11
		Python	Opportunistic	7	0	0	0	5
	URL	**Java**	Opportunistic	11	0	0	0	2
		Python	-	-	-	-	-	-
	I/O	**Java**	Opportunistic	24	6	2	1	1
		Python	-	-	-	-	-	-
Constant Description	**REGEX**	**Java**	Systematic	34	0	0	1	0
		Python	Opportunistic	15	1	3	7	10
	URL	**Java**	-	-	-	-	-	-
		Python	-	-	-	-	-	-
	I/O	**Java**	Opportunistic	23	12	4	0	7
		Python	-	-	-	-	-	-
Input Configuration Options	**REGEX**	**Java**	Systematic	6	0	0	0	0
		Python	Opportunistic	14	13	3	12	-
	URL	**Java**	-	-	-	-	-	-
		Python	-	-	-	-	-	-
	I/O	**Java**	Opportunistic	4	2	0	0	2
		Python	-	-	-	-	-	-

7

Conclusion

Our case study on the instructional documentation of `REGEX`, `URL` and `I/O` in Java and Python revealed that there exists patterns of reusing information from reference documentation. We found that between 45-76% of instructional documentation was *related to API*. We further mapped sentences in the instructional documentation to their possible source, i.e. the API reference documentation. As a result of this process, we identified six types of sentence matches, namely *exact*, *manipulated*, *rephrased*, *partial*, *inferred* and *no match*. The percentage of occurrence of these match types varies between Java and Python as well as among the API topics, but reveal interesting observations such as the existence of redundancies and inconsistencies in information. We also discovered that the most common reasons for *no matches* are because they describe the *underlying topic*, *usage*, *internal working*, or *behaviour* information. We found a total of 38 instances of documentation reuse that follow certain patterns. We categorized these patterns into four types based on their purpose and structure, i.e. sentences from the *Class Description*, *Method Description*, *Constant Description* or *Input Configuration Options*. We also measured the *impact* on the existing instructional documentation if these patterns were to be applied automatically using a set of proposed parameters. In general, the automation of the *systematic* pattern instances, i.e. ones in which the pattern is consistently used, tends to involve less change in documentation as opposed to *opportunistic*, less consistent instances.

This work provides insight into how information is and can be reused across documentation types. It is inclined towards promoting consistent information across different

documentation types of software development tools. Additionally, it intends to support the automation of documentation reuse to reduce the time and effort taken by documentation authors. We performed our exploratory study on Java and Python to campaign towards generalizability across different software programming languages and their APIs. This would allow both readers and authors to move between different languages and their documentations with greater ease. While prior work has focused on different aspects of documentation duplication detection and its removal, we hope to instead provide the foundation for a deeper understanding of documentation reuse and its support.

As the first step towards a better understanding of current documentation and the relationship between different documentation types in terms of information reuse, a number of future avenues unfold.

Wider exploratory analysis into the different match types could reveal further interesting and useful characteristics. Some characteristics like redundancies and inconsistencies could be used to alert documentation creators of the unintentional properties that arise, as well as to point out favorable text structures. This could help bridge the gap between documentation and the information needs of readers. Further, we leave the analysis of the *inferred* matches to future work with the intention of involving multiple annotators of different expertise levels in order to annotate the inferential ability of instructional documentation sentences from API reference sentences.

Our work studied the number of sentences impacted in the instructional documentation if the reuse patterns were to be incorporated automatically. However understanding the outcome of automation at the document level, such as the impact on the size and coherence of the documentation as a whole, would provide greater insight and motivation for automation.

We aim to pursue a documentation creation aide tool to formalize the reuse patterns elicited in this work as well as new patterns that may be observed. Such a tool would mitigate the effort put in by authors in the documentation creation process, potentially improve the quality of the documentation and provide familiarity when reading documentation of different APIs in different programming languages.

References

Below is the list of web URLs referenced in this thesis. In the case of snippets of documentation used as examples, the corresponding URL defines the particular file in which the example text can be found.

[1] docs.oracle.com/javase/8/docs/

[2] docs.python.org/3/

[3] docs.oracle.com/javase/8/docs/api/java/util/regex/package-summary.html

[4] www.stackoverflow.com

[5] docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html

[6] docs.python.org/3/library/socket.html

[7] docs.oracle.com/javase/8/docs/api/java/util/regex/package-summary.html

[8] docs.oracle.com/javase/8/docs/api/java/net/package-summary.html

[9] docs.oracle.com/javase/8/docs/api/java/nio/file/package-summary.html

[10] docs.oracle.com/javase/tutorial/essential/regex/

[11] docs.oracle.com/javase/tutorial/networking/urls/index.html

[12] docs.oracle.com/javase/tutorial/essential/io/fileio.html

[13] docs.python.org/3/library/re.html

[14] docs.python.org/3/library/urllib.html

[15] docs.python.org/3/library/functions.html

[16] docs.python.org/3/howto/regex.html

[17] docs.python.org/3/howto/urllib2.html

[18] docs.python.org/3/tutorial/inputoutput.html

[19] docs.oracle.com/javase/tutorial/essential/io/fileAttr.html

[20] docs.oracle.com/javase/tutorial/essential/regex/pattern.html

[21] docs.oracle.com/javase/tutorial/essential/regex/intro.html

[22] docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

[23] docs.python.org/3/library/urllib.request.html

[24] docs.oracle.com/javase/tutorial/essential/regex/matcher.html

[25] docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html

[26] docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html

[27] docs.oracle.com/javase/tutorial/networking/urls/creatingUrls.html

[28] docs.oracle.com/javase/8/docs/api/java/net/URL.html

[29] docs.oracle.com/javase/tutorial/essential/io/rafs.html

[30] docs.oracle.com/javase/8/docs/api/java/nio/channels/SeekableByteChannel.html

[31] docs.oracle.com/javase/tutorial/essential/io/file.html

[32] docs.python.org/3/library/json.html

[33] docs.oracle.com/javase/tutorial/networking/urls/readingURL.html

[34] docs.oracle.com/javase/tutorial/essential/io/pathOps.html

[35] docs.oracle.com/javase/tutorial/essential/regex/unicode.html

[36] docs.oracle.com/javase/tutorial/essential/io/pathClass.html

[37] docs.oracle.com/javase/tutorial/essential/io/walk.html

[38] docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitor.html

[39] docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/BasicFileAttributeView.html

[40] docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/AclFileAttributeView.html

[41] docs.oracle.com/javase/tutorial/networking/urls/urlInfo.html

[42] docs.oracle.com/javase/tutorial/essential/io/move.html

[43] docs.oracle.com/javase/8/docs/api/java/nio/file/StandardCopyOption.html

[44] nltk.org

[45] spacy.io

[46] docs.oracle.com/javase/tutorial/essential/regex/groups.html

[47] devguide.python.org/documenting

[48] docs.oracle.com/javase/tutorial/essential/io/formatting.html

[49] docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitResult.html

Bibliography

[1] Palakorn Achananuparp, Xiaohua Hu, and Xiajiong Shen. The Evaluation of Sentence Similarity Measures. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery*, pages 305–316, 2008.

[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*, pages 1199–1210, 2019.

[3] Ademar Aguiar and Gabriel David. Patterns for Documenting Frameworks–Part I. *Proceedings of VikingPLoP*, 2005.

[4] Ademar Aguiar and Gabriel David. Patterns for Documenting Frameworks: Customization. In *Proceedings of the Conference on Pattern Languages of Programs*, page 16, 2006.

[5] Ademar Aguiar and Gabriel David. Patterns for Effectively Documenting Frameworks. In *Transactions on Pattern Languages of Programming II*, pages 79–124. 2011.

[6] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[7] Gianni Angelini. Current Practices in Web API Documentation. In *European Academic Colloquium on Technical Communication*, page 70, 2018.

[8] Greg Butler, Peter Grogono, and Ferhat Khendek. A Reuse Case Perspective on Documenting Frameworks. In *Proceedings of Asia Pacific Software Engineering Conference*, pages 94–101, 1998.

[9] Gregory Butler, Rudolf K. Keller, and Hafedh Mili. A Framework for Framework Documentation. *ACM Computing Surveys*, 2000.

[10] 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*, pages 127–136, 2010.

[11] Uri Dekel and James D Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 320–330, 2009.

[12] Joseph L Fleiss. Measuring Nominal Scale Agreement Among Many Raters. In *Psychological Bulletin*, volume 76, page 378, 1971.

[13] Andrew Forward and Timothy C Lethbridge. The Relevance of Software Documentation, Tools and Technologies: A Survey. In *Proceedings of the ACM Symposium on Document Engineering*, pages 26–33, 2002.

[14] Adam Fourney and Michael Terry. Mining Online Software Tutorials: Challenges and Open Problems. In *Proceedings of Extended Abstracts on Human Factors in Computing Systems*, pages 653–664, 2014.

[15] Golara Garousi, Vahid Garousi, Mahmoud Moussavi, Guenther Ruhe, and Brian Smith. Evaluating Usage and Quality of Technical Software Documentation: An Empirical Study. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, pages 24–35, 2013.

[16] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. An Unsupervised Approach for Discovering Relevant Tutorial Fragments for APIs. In *Proceedings of the 39th International Conference on Software Engineering*, pages 38–48, 2017.

[17] Ralph E Johnson. Documenting Frameworks Using Patterns. In *Proceedings of Object-oriented Programming, Systems, Languages, and Applications*, volume 92, pages 63–76, 1992.

[18] Dmitrij Koznov, Dmitry Luciv, Hamid Abdul Basit, Ouh Eng Lieh, and Mikhail Smirnov. Clone Detection in Reuse of Software Technical Documentation. In *Proceedings of International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 170–185, 2015.

[19] D.V. Koznov, D.V. Luciv, and G.A. Chernishev. Duplicate Management in Software Documentation Maintenance. In *Proceedings of the 5th International Conference on Actual Problems of System and Software Engineering. CEUR Workshops proceedings*, volume 1989, pages 195–201, 2017.

[20] Douglas Kramer. API Documentation from Source Code Comments: A Case Study of Javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation*, pages 147–153, 1999.

[21] Klaus Krippendorff. *Content Analysis: An Introduction to its Methodology*. Sage Publications, 2018.

[22] D.V. Luciv, D.V. Koznov, H.A. Basit, and A.N. Terekhov. On Fuzzy Repetitions Detection in Documentation Reuse. In *Programming and Computer Software*, volume 42, pages 216–224, 2016.

[23] D.V. Luciv, D.V. Koznov, G.A. Chernishev, A.N. Terekhov, K. Yu. Romanovsky, and D.A. Grigoriev. Detecting Near Duplicates in Software Documentation. In *Programming and Computer Software*, volume 44, pages 335–343, 2018.

[24] Walid Maalej and Martin P. Robillard. Patterns of Knowledge in API Reference Documentation. In *IEEE Transactions on Software Engineering*, volume 39, pages 1264–1282, 2013.

[25] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. Application Programming Interface Documentation: What do Software Developers Want? In *Journal of Technical Writing and Communication*, volume 48, pages 295–330, 2018.

[26] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. How Developers use API Documentation: An Observation Study. In *Communication Design Quarterly Review*, volume 7, pages 40–49, 2019.

[27] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What Should Developers be Aware of? An Empirical Study on the Directives of API Documentation. In *Empirical Software Engineering*, volume 17, pages 703–737, 2012.

[28] Mohamed A Oumaziz, Alan Charpentier, Jean-Rémy Falleri, and Xavier Blanc. Documentation Reuse: Hot or Not? An Empirical Study. In *Proceedings of International Conference on Software Reuse*, pages 12–27, 2017.

[29] Chris Parnin and Christoph Treude. Measuring API Documentation on the Web. In *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*, pages 25–30, 2011.

[30] Vir Phoha. A Standard for Software Documentation. In *Computer*, volume 30, pages 97–98, 1997.

[31] Jitendra Rama Aswadh Josyula and Soma Sekhara Sarat Chandra Panamgipalli. *Identifying the Information Needs and Sources of Software Practitioners: A Mixed Method Approach.* Master's thesis, 2016.

[32] R. Ries. IEEE Standard for Software User Documentation. In *International Conference on Professional Communication, Communication Across the Sea: North American and European Practices*, pages 66–68, 1990.

[33] Martin P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. In *IEEE software*, volume 26, pages 27–34, 2009.

[34] Martin P. Robillard and Robert Deline. A Field Study of API Learning Obstacles. In *Empirical Software Engineering*, volume 16, pages 703–732, 2011.

[35] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, 2012.

[36] Chandan R Rupakheti. *A Critic for API Client Code using Symbolic Execution*. PhD thesis, Clarkson University, 2012.

[37] Ian Sommerville. Software Documentation. In *Software Engineering*, volume 2, pages 143–154, 2001.

[38] Christoph Treude and Martin P. Robillard. Augmenting API Documentation with Insights from Stack Overflow. In *Proceedings of 38th International Conference on Software Engineering*, pages 392–403, 2016.

[39] Christoph Treude, Martin P. Robillard, and Barthélémy Dagenais. Extracting Development Tasks to Navigate Software Documentation. In *IEEE Transactions on Software Engineering*, volume 41, pages 565–581, 2014.

[40] Gias Uddin and Martin P. Robillard. How API Documentation Fails. In *IEEE Software*, volume 32, pages 68–75, 2015.

[41] Robert Watson, Mark Stamnes, Jacob Jeannot-Schroeder, and Jan H Spyridakis. API Documentation and Software Community Values: A Survey of Open-source API Documentation. In *Proceedings of the 31st ACM International Conference on Design of Communication*, pages 165–174, 2013.

[42] Robert B Watson. Development and Application of a Heuristic to Assess Trends in API Documentation. In *Proceedings of the 30th ACM International Conference on Design of Communication*, pages 295–302, 2012.

[43] Sven Wildermann. *Messung der Informationstypen-Häufigkeiten in der Python-Dokumentation.* Bachelor's thesis, 2014.

[44] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring Resource Specifications from Natural Language API Documentation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 307–318, 2009.

Appendix

A Issues in Preprocessing

A few of the issues faced during preprocessing of the data are presented below:

- Though all the files are in HTML format, the structure of the HTML vary across the two development tools.
- Items in unordered and ordered lists do not always contain punctuation to indicate the end of a sentence. In some scenarios, list items remain incomplete, with the following item continuing the previous item with connectors such as *or ...* or *and ....* Hence, it was difficult to decide whether list items were to be merged or to be treated as individual sentences.
- When sentences contain 'For e.g. X', if split on typical punctuation such as . (a period followed by a space), this would be split into two sentences: 'For e.g.' and 'X'.
- Sentence tokenizers from natural language libraries such as $SpaCy$ and $nltk$ would often perform unfavorable sentence splits such as splitting 'Pattern.CANON_EQ' into 'Pattern', '.' and 'CANON_EQ' or splitting $.*[.](?!bat\$)[.]*\$$ into $.*[.](?!bat\$)[.$ and $]*\$$ respectively. This was problematic as key characters, such as those in regular expressions, would be treated as natural language punctuation, incorrectly.

B Preprocessing Steps

In general, the following rules and preprocessing techniques for sentence extraction were adhered to:

- Remove HTML tags *script*, *style*, *table*
- Insert a comma after the tokens 'e.g.' and 'i.e.'

- Insert a comma after the token ‘etc.’ if the word following this one began with a lower case alphabet.
- Replace multiple adjacent commas (occurring as a result of previous preprocessing steps) with a single comma.
- Replace newlines with spaces
- Replace multiple adjacent spaces with a single space
- Replace multiple adjacent periods (...) with a single period (.)
- In general, blockquotes, code blocks, images and the equivalents across the files were replaced by a single token *BLOCKQUOTE*, *CODE* and *IMAGE* respectively. These blocks were identified as being of a specific HTML tag type or having a specific HTML class.
- If a list item did not end in a period, the following item would be concatenated to the previous, separated by a semicolon.
- Finally, split on on a period followed by a space (‘. ’) and an exclamation followed by a space (‘! ’) to produce individual sentences

It is important to note here that inline HTML code tags in the sentence (*inline* and hence, did not involve line breaks) were maintained as is. Usually such pieces were names of the library or method being described. For example, ’The *java.util.regex* package primarily consists of three classes: *Pattern*, *Matcher*, and *PatternSyntaxException*.‘.

C List of Unique Source and Destination Contexts

Below is the lists of source and destination contexts annotated using open-coding as the structure in which a sentence lies in the source API reference documentation or the destination instructional documentation.

C.1 Contexts for API Reference Documentation Sentences

These context values refer to the sentence context largely in terms of position in the API reference documentation and the extent that has been copied to the instructional documentation. The following are the unique contexts annotated for matched sentences in the API

reference documentation of `REGEX`, `URL` and `I/O` in Java and Python.

- Sentence in language guide
- Definition fragment from leading sentence in class
- Topic/Concept description
- Catalog of input configuration options
- Leading sentence in module/package description
- Sentence in module/package description
- Leading sentence in class description
- Sentence in class description
- Leading sentence in constant description
- Sentence in constant description
- Entire constant description
- Method definition
- Leading sentence in method description
- Sentence in method description
- Leading paragraph of method description
- 2nd paragraph of method description
- *Parameters:* in method description
- *Returns:* in method description
- Entire method description
- Elaboration of examples of two methods with similar functionality
- Comparison with other language
- Notes and Warnings

C.2 Contexts for Instructional Documentation Sentences

These context values refer to the sentence context largely in terms of the structure in which the sentence lies in the instructional documentation. The following are the unique contexts

annotated for matched sentences in the instructional documentation of the `REGEX`, `URL` and `I/O` in Java and Python.

- Introduction
- API topic representation
- Input configuration options
- Class introduction and description
- Catalog of classes in package/module
- Inline/embedded method reference documentation
- Catalog of class methods
- Catalog with methods of related functionality
- Catalog with methods of different API related to this one
- Catalog with methods with same parameters (based on same concept)
- Catalog of "other useful" methods
- Catalog of constants
- Topic with examples
- Usage with examples
- Use-case and solution
- Limitations and deprecation information
- Notes and Warnings