A Qualitative Study: How *Solution Snippets* are Presented in Stack Overflow and How those *Solution Snippets* Need to be Adapted for Reuse

by

Nimmi R. Weeraddana
B.Sc., University of Moratuwa, Sri Lanka, 2018

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

A Qualitative Study: How *Solution Snippets* are Presented in Stack Overflow and How those *Solution Snippets* Need to be Adapted for Reuse

by

Nimmi R. Weeraddana
B.Sc., University of Moratuwa, Sri Lanka, 2018

Supervisory Committee

Dr. Daniel M. German, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Departmental Member
(Department of Computer Science)

ABSTRACT

Researchers use datasets of Question-Solution pairs to train machine learning models, such as source code generation models. A Question-Solution pair contains two parts: a programming question and its corresponding *Solution Snippet*. A *Solution Snippet* is a source code that solves a programming question. These datasets of Question-Solution pairs can be extracted from a number of different platforms. In this research, I study how Question-Solution pairs are extracted from Stack Overflow (SO). There are two limitations of datasets of Question-Solution pairs extracted from SO: (1) according to the authors of these datasets, some Question-Solution pairs contain *Solution Snippets* that do not solve the question correctly, and (2) these datasets do not contain the information on how *Solution Snippets* need to be reused, and such information would enhance the reusability of *Solution Snippets*. These limitations of datasets of pairs could adversely affect the quality of the code being generated by machine learning models. In this research, I conducted a qualitative study to categorize various presentations of *Solution Snippets* in SO's answers as well as how *Solution Snippets* can be adapted for reuse. By doing so, I identified eight categories of how *Solution Snippets* are presented in SO's answers and five categories of how *Solution Snippets* could be adapted. Based on these results, I concluded several potential reasons why it is not easy to create datasets of Question-Solution pairs. The first categorization informs that finding the correct location of the *Solution Snippet* is challenging when there are several code blocks within the answer to the question. Subsequently, the researcher must identify which code within that code block is the *Solution Snippet*. The second categorization informs that most *Solution Snippets* appear challenging to be adapted for reuse, and how *Solution Snippets* are potentially adapted is not explicitly stated in them. These insights shed light on creating better quality datasets from questions and answers posted on Stack Overflow.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables vii

List of Figures viii

Acknowledgements xi

Dedication xii

1 Introduction 1

1.1 **Motivation** . . . 2
1.2 **Problem** . . . 3
1.3 **Goal and Research Questions** . . . 5
1.4 **Research Design and Methodology** . . . 6
1.5 **Results and Contributions** . . . 7
1.5.1 **A categorization of how solution snippets are presented in answers** . . . 7
1.5.2 **A categorization of the reusability aspects of solution snippets** . . . 8
1.5.3 **An analysis of the above two categorizations.** . . . 8
1.5.4 **A manually annotated dataset** . . . 9
1.6 **Thesis Outline** . . . 10

2 Background 11

2.1 ***Solution Snippets*** . . . 11

2.2 **Datasets of Question-Solution pairs that were extracted from Stack Overflow and their limitations** . . . 13
2.3 **Datasets of Question-Solution pairs that were extracted from non-Stack Overflow sources and their limitations** . . . 17
2.4 **Categorizations of Stack Overflow's questions and answers, and their limitations** . . . 18

3 The Approach **21**
3.1 **Research Goal and Research Questions** . . . 21
3.2 **The Philosophical Worldview of this Study** . . . 22
3.3 **Research Design** . . . 23
3.3.1 **Data Collection** . . . 24
3.3.2 **Identifying *how-to* questions which are likely to contain *Solution Snippets* in their answers** . . . 24
3.3.3 **Coding the answers to *how-to* questions** . . . 27
3.4 **Researcher's Role** . . . 30
3.5 **Summary** . . . 30

4 Findings of the Study **31**
4.1 **RQ1. How are solution snippets presented in Stack Overflow's answers?** . . . 31
4.1.1 **The solution snippet is inlined with the text.** . . . 31
4.1.2 **The code block contains only the solution snippet.** . . . 33
4.1.3 **The answer contains more than one solution snippet.** . . . 33
4.1.4 **The code block that contains the solution snippet also contains sample inputs/outputs:** . . . 34
4.1.5 **The code block that contains the solution snippet also contains inputs and code snippets from the question's body.** . . . 35
4.1.6 **The solution snippet is an example of a programming concept.** . . . 35
4.1.7 **The answer does not contain solution snippets.** . . . 36
4.1.8 **The answer contains solution snippets for related problems.** . . . 37

4.2 **RQ2. How are solution snippets potentially adapted in order to be reused?** . . . 38
4.2.1 **As-is** . . . 38
4.2.2 **User-defined functions (UDF)** . . . 38
4.2.3 **Placeholders** . . . 40
4.2.4 **Customizable variables and/or data** . . . 43
4.2.5 **Conceptual** . . . 43
4.3 **Summary** . . . 44

5 Comparison to Other Research **46**

6 Discussion **48**
6.1 **Observations made about Stack Overflow's questions** . . . 48
6.2 **Why is it challenging for automatic extractors to find and extract *Solution Snippets*?** . . . 49
6.2.1 **It is not easy to find *Solution Snippets*.** . . . 50
6.2.2 **It is challenging to separate a *Solution Snippet* from the ancillary code within a code block.** . . . 54
6.3 **What are the potential difficulties of adapting Stack Overflow's *Solution Snippets*?** . . . 57
6.4 **What Stack Overflow could do to help improve the automatic extraction of *Solution Snippets*.** . . . 61
6.5 **Threats to Validity** . . . 62
6.6 **Summary** . . . 64

7 Conclusions and Future Work **66**
7.1 **Conclusions** . . . 66
7.2 **Future Work** . . . 67

Bibliography **68**

A The questions downloaded from the SOTorrent database **74**

B How the categories identified in my study evolved **76**

C The manually coded dataset of my study **79**

List of Tables

Table 1.1 Methods used to present the solution snippet in an answer (the answer to RQ1). 7
Table 1.2 Categories for the reusability of solution snippets (the answer to RQ2). 8

Table 2.1 Limitations of existing datasets of Question-Solution pairs created from Stack Overflow 15
Table 2.2 Existing categorizations and limitations with respect to locating *Solution Snippets* in answers and the reusability of *Solution Snippets* 20

Table 3.1 The number of questions classified, the number of *how-to* questions, the number of coders, the agreement at the end of iteration 26
Table 3.2 Total number of questions and answers studied 29

Table 4.1 The answer to RQ1. How are *Solution Snippets* presented in Stack Overflow's answers? 32
Table 4.2 The answer to RQ1: How *Solution Snippets* are potentially adapted in order to be reused 39

Table A.1 A sample dataset used to prepare the Google Sheets for the first part of my study 75

Table B.1 Total number of questions and answers studied 78

List of Figures

Figure 1.1 An answer to the question *"How can I check if a key exists in a dictionary?"* 2

Figure 2.1 An excerpt of an accepted answer showing its first two code blocks. Each of them contains a *Solution Snippet* that is surrounded by ancillary code. 12
Figure 2.2 An excerpt of an answer having a *Solution Snippet* with a variable named `nodesArray` that needs to be replaced when reusing. . . 13

Figure 3.1 An overview of the research design 24
Figure 3.2 The process I used to identify *how-to* questions. 25
Figure 3.3 The methodology used to study Stack Overflow's answers and *solution snippets*. 27

Figure 4.1 An answer with an inline *solution snippet* (annotated in red). . 33
Figure 4.2 An answer containing only one code block in which the *solution snippet* spans the entire code block. 33
Figure 4.3 An answer with several blocks containing a *solution snippet*: the first two are specific to different JavaScript versions, and the next two use different libraries. 34
Figure 4.4 A code block with a *solution snippet*, its input, and its outputs. 35
Figure 4.5 The body of the question *"push multiple elements to array."* . . 36
Figure 4.6 The accepted answer to the question in Figure 4.5 that reuses the same variable `a` and values `[1,2]`. 36
Figure 4.7 A *solution snippet* that is an example. 36
Figure 4.8 An answer that explains how to solve the question without providing a *solution snippet*. 37
Figure 4.9 The body of the question *"How can I get last characters of a string?"* 37

Figure 4.10The *solution snippet* that solves the problem *"getting the characters after the underscore in a string"* which was not asked in the original question (in Figure 4.9. 38
Figure 4.11An *as-is solution snippet* which requires no customization in order to be reused. 38
Figure 4.12A *UDF*. The reuse of this snippet requires making a call to this UDF — `getKeys`. 40
Figure 4.14A *Solution Snippet* that contains print statements as placeholders. 41
Figure 4.16Using angle brackets (<>) to encapsulate the text that indicates that it needs to be customized. 41
Figure 4.20A *solution snippet* with customizable variables and/or data. . . 43
Figure 4.21A Conceptual *solution snippet*. 44

Figure 6.1 The first three code blocks of an answer to a Python question. The *Solution Snippets* are found in the first two code blocks. . . 51
Figure 6.2 An answer to a Python question. This answer contains an inline code snippet as well as a code block. The inline code snippet is a *Solution Snippet*, and the code block contains an example and its output. 52
Figure 6.3 An answer to a Java question. It contains two code blocks, and one *Solution Snippet* spans over these two code blocks. 52
Figure 6.4 An answer to a Javascript question without Javascript source code in its code block. The answer contains a code block and an inline code snippet, but both are in CSS. 53
Figure 6.5 A code block that contains a prologue, a *solution snippet*, and an epilogue. 55
Figure 6.6 A code block that contains a much simpler prologue that establishes the context for the *Solution Snippet* in use. 55
Figure 6.7 A code block that contains two *solution snippets*. The first *solution snippet* is formed by snippets (a) and (b), and the second *solution snippet* is formed by non-contiguous snippets (a) and (c). 56
Figure 6.8 The body of the question *"How to get [the cuurent] Month [using] JavaScript in 2 digit format?"* 58
Figure 6.9 A *Solution Snippet* containing a reference (`this`) to an object that is not defined within any code block in the answer. 58

Figure 6.10A code block in a question 59
Figure 6.11A *Solution Snippet* having an array that is not defined within any code block in the answer. 59
Figure 6.12A *Solution Snippet* that has two placeholders. 59
Figure 6.13A *Solution Snippet* having a print statement that is a placeholder. This particular placeholder encloses a part of the *Solution Snippet* (`index++`). 60
Figure 6.14A print statement that is part of the solution to the question . 60
Figure 6.15A *Solution Snippet* that has two sections: an import statement and the remaining source code in the *Solution Snippet*. 61
Figure 6.16A conceptual *Solution Snippet*. 61

ACKNOWLEDGEMENTS

I would like to thank all my university friends for helping me have a good university life, especially during the hard times due to the global pandemic, and I would like to express my special gratitude to:

My husband, Sanka Weerathunga, for his immense love and support.

My supervisor, Dr. Daniel German, for giving me the opportunity to pursue post-graduate studies and for the support, encouragement, helping me develop my research skills.

Dr. Margaret-Anne Storey and Dr. Sebastian Baltes, for their support and encouragement throughout my research.

My parents, Sumanasena and Sakunthala, my brother, Chathuranga, and my sister, Madhu, for their endless love and courage.

My best friend, Cassandra Cupryk, for being my project partner on several graduate projects and for cheering me up whenever I most needed it.

All the department members, for the support.

DEDICATION

I am dedicating this thesis to my parents, Sumanasena and Sakunthala, for their immense love and encouragement.

1

Introduction

Machine learning models for source code generation[1] ([4, 16, 35, 36, 42]), querying source code snippets[2] ([39, 52]), and source code summarization[3] ([28]) need to be trained using pairs of programming problems and their corresponding source code that solve them. Such pairs are called *Question-Solution pairs* in this thesis. A Question-Solution pair has two parts: (1) a *natural language* text that explains a programming question and (2) the source code that solves it (throughout this thesis, I call the source code that solves a programming question a *Solution Snippet*). A good Question-Solution pair is one in which the question is succinct and clearly stated, and its *Solution Snippet* precisely answers the question [54]. In the context of this thesis, a dataset is of high quality if it is composed of good pairs. To create better machine learning models, such high-quality training datasets are needed [54]. Otherwise, the errors in the training dataset would propagate to the machine learning models that are being generated [38].

In order to create datasets of Question-Solution pairs, researchers often use Stack Overflow[4], a social media Q&A platform, because it contains programming questions and *Solution Snippets* in one place [4, 28, 51, 53, 54]. They have created datasets of Question-Solution pairs by pairing the title of a question with a *Solution Snippet* in one of its answers. Usually, the title of a Stack Overflow's question precisely summarizes the programming problem [47]. Once the title is extracted, the researchers

[1] A source code generation model aims to generate a code given a natural language description [4].

[2] A code search model measures the similarity of semantics between a natural language description and a collection of source code to find the most relevant source code in the collection [52].

[3] A source code summarization model aims to generate a natural language description (a comment) for a given piece of source code [28].

[4] `https://stackoverflow.com/`

have to extract the *Solution Snippets*. Because *Solution Snippets* are embedded within the text and code blocks in answers, it is challenging to extract *Solution Snippets* to create good Question-Solution pairs; I will explain why in the following two sections.

1.1 Motivation

For this study, I used Stack Overflow's questions that are likely to be answered with *Solution Snippets*. Such questions are called *how-to* questions in this thesis. A *how-to* question provides a scenario and asks how to implement it [34]. Several other researchers have also identified *how-to* questions as a key category of Stack Overflow's questions [12, 13, 46, 55]. For example, the question *"How can I check if a key exists in a dictionary?"* (a Python question) is a *how-to* question, and it is expected to contain *Solution Snippets* in its answers.

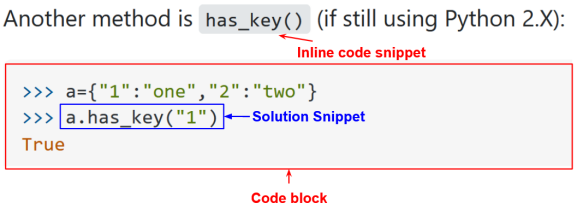


Figure 1.1: An answer to the question *"How can I check if a key exists in a dictionary?"*

Figure 1.1 shows the second top-scored answer to the above question. As shown in this figure, source code[5] in answers to Stack Overflow's questions is typically found within code blocks and code snippets inline with the text. In this answer, the *Solution Snippet* (annotated in blue) is found within the code block and does not span the entire code block. To create a Question-Solution pair, researchers could pair the title of the above question with this *Solution Snippet*. The rest of the source code within the code block shows how to initialize the variable `a` (used in the *Solution Snippet*) with a sample input (`{"1":"one","2":"two"}`) and the corresponding output of the *Solution Snippet*. Even though this extra source code is not part of the *Solution Snippet*, it contains contextual information about the reusability of this *Solution Snippet*—how the variable `a` needs to be initialized when reusing. Because such contextual

[5]Source code in Stack Overflow is enclosed with `<code>` tags in the HTML markup. If a `<code>` tag is enclosed within a `<p>` (paragraph) tag, it will be rendered as a small grey code snippet in line with the text on the website; otherwise, it will be rendered as a large grey section (a code block) that is separated from the text.

information is potentially useful to train machine learning models to generate source code, the researchers might have to extract extra source code and *Solution Snippets* within code blocks separately when extracting datasets of Question-Solution pairs from Stack Overflow.

On the other hand, the question *"What is the difference between a function expression vs declaration in JavaScript?"* is not a *how-to* question. It requires the answers to provide an explanation, and it is not expected to contain *Solution Snippets* in its answers. Thus, a Question-Solution pair cannot be extracted from this question and its answers.

One way to create a dataset of Question-Solution pairs from Stack Overflow is by manual extraction, which usually yields high-quality datasets [31, 50], but it is a laborious task. Because the manually extracted datasets often contain a small number of pairs, they may not be sufficient to train sophisticated machine learning models [31, 50].

The other way to create a dataset of Question-Solution pairs is by automatic extraction. The existing automated methods ([4, 28, 53, 54]) have created large datasets, but those are of poor quality [31, 50]. Some authors have paired questions' titles to the entire code blocks in their answers, assuming that *Solution Snippets* always span over entire code blocks [4, 28, 53]. Also, the authors of [53, 54] have acknowledged that some pairs in their datasets are incorrect.

1.2 Problem

The overarching research problem of this study is:

> "It is challenging to automatically extract high-quality datasets of Question-Solution pairs from Stack Overflow."

The following are the requirements of a high-quality dataset of Question-Solution pairs and any corresponding challenges.

The natural language text of a Question-Solution pair should correctly describe a programming problem. Authors of Stack Overflow's questions take reasonable care to create titles that summarize problems (in natural language), and it is easy to extract the problem from the title of a question [28, 42, 47].

The *Solution Snippet* of a Question-Solution pair should correctly solve the programming problem [54]. When manually curating datasets, researchers are able to identify and extract *Solution Snippets* from the rest of the source code within code blocks, and this requirement is usually satisfied [31, 51]. In automatic extraction, it is challenging to extract correct *Solution Snippets* from answers, and some *Solution Snippets* extracted by existing methods are known to be incorrect [31, 53, 54].

The information on how *Solution Snippets* need to be adapted for reuse should be documented. It is not easy to identify how to adapt *Solution Snippets* for reuse (whether to change the variables, data, or parameters, among others). Such information often has not been considered when creating the existing datasets of Question-Solution pairs.

The number of Question-Solution pairs should be large enough to train some machine learning models [50]. Since Stack Overflow claims[6] to have over 22 million programming questions and 32 million answers as of November 2021, it is a rich source to extract Question-Solution pairs. Datasets created by automated methods are large and often scalable [31, 50]. However, manually curated datasets are not large enough to train some machine learning models [31, 50].

In summary, researchers require large, high-quality datasets of Question-Solution pairs to train some machine learning models, but it is challenging to create such datasets. There is a trade-off between the number of Question-Solution pairs and the quality of their *Solution Snippets*. In addition, identifying the reusability aspect of *Solution Snippets* is also not easy.

[6] `https://stackexchange.com/sites`

1.3 Goal and Research Questions

The overarching goal of this study is to:

> Understand how *Solution Snippets* are presented in Stack Overflow's answers and how those *Solution Snippets* need to be adapted for reuse (specifically for the purpose of creating datasets of Question-Solution pairs to train machine learning models).

The expected beneficiaries of this study are the researchers who create automatic extractors for creating datasets of Question-Solution pairs from Stack Overflow. When extracting *Solution Snippets* to create such datasets, the researchers may require answers to several questions, including the following: Where are *Solution Snippets* located in Stack Overflow's answers (within code blocks and/or inline with the text)? If a *Solution Snippet* appears within a code block, does it spans the entire code block? If not, what is extra source code within code blocks? Are there multiple *Solution Snippets* within a code block? The following research question of this study aims to give researchers the understanding to answer the above questions.

> **RQ1. How are *Solution Snippets* presented in Stack Overflow's answers?**

Once how *Solution Snippets* are found in Stack Overflow's answers is identified, the researchers have to understand how *Solution Snippets* need to be adapted for reuse. In a *Solution Snippet*, there could be different parts that have to be adapted for reuse. If so, what are those parts? How do those parts need to be changed? Are there any *Solution Snippets* that can be reused as they are? The objective of the next research question of this study is to find answers to the above questions.

> **RQ2. How are *Solution Snippets* potentially adapted in order to be reused?**

The answers to the above two research questions give researchers an understanding of what are *Solution Snippets* and how they could be potentially reused. This understanding would help researchers push the boundaries of the state-of-the-art automatic extractors designed to extract *Solution Snippets* to create datasets of Question-Solution pairs from Stack Overflow.

1.4 Research Design and Methodology

This study takes the *constructivist worldview* to design its methodology. The constructivist worldview is focused on "understanding the world and making sense of the meanings that others have about the world" [17]. It is suitable when a research problem is broadly defined and when research questions are exploratory. In this study, the research questions aim to explore how *Solution Snippets* are presented in answers and how they are potentially adapted for reuse; thus, the research questions are exploratory. Having such research questions in this study matches the *constructivist worldview*.

"The *constructivist worldview* is typically seen as an approach to ***qualitative research***" [17]. Unlike quantitative research methods designed to measure variables and test hypotheses, qualitative research methods are designed to understand a phenomenon, a concept, etc. Thus, researchers often code the data to create categories, themes, and theories in qualitative research [17]. Hence, I designed a qualitative study as outlined below. There are two main parts of this study:

1. Identifying a collection of *how-to* questions which are likely to contain *Solution Snippets* in their answers.

2. Coding the answers to these *how-to* questions and creating two categorizations, one for each research question.

For this study, Stack Overflow's questions and answers of the three most popular[7] programming languages were used: *JavaScript*, *Java*, and *Python*. I obtained the top-1000 questions for each language from SOTorrent [8] database, ranked by the score[8] in descending order. In the first part of this study, three coders (including myself) classified batches of those questions until 100 *how-to* questions were identified for each language (300 *how-to* questions in total).

For the second part of the study, the coders coded *Solution Snippets* (in the answers) to the *how-to* questions until they reached the saturation of the categories. In Section 3.3, I will provide a detailed description of the coding process followed in order to identify the categories.

[7]https://stackoverflow.com/tags

[8]The score of a question or an answer on Stack Overflow is the number of upvotes minus downvotes for it. The score indicates the popularity of a question or an answer.

1.5 Results and Contributions

This qualitative study resulted in two categorizations, one for each research question: (1) a categorization of how *Solution Snippets* are presented in answers and (2) a categorization of the reusability aspects of *Solution Snippets*. By analyzing these categorizations, I identified potential reasons why it is challenging to extract *Solution Snippets* to create high-quality datasets. Lastly, the manually annotated dataset of this study is made available for reproducibility and traceability. I describe these contributions in detail below.

1.5.1 A categorization of how solution snippets are presented in answers

This is the answer to RQ1. It has eight categories, and they are briefly described in Table 1.1.

Table 1.1: Methods used to present the solution snippet in an answer (the answer to RQ1).

Categories	**Description**
The *Solution Snippet* is inlined with the text	The *Solution Snippet* is not in a code block in the answer, but appear inline with *natural language* text in the answer.
The code block contains only the *Solution Snippet*	The answer contains only one code block and, the *Solution Snippet* spans this entire code block.
The answer contains more than one *Solution Snippet*	The answer contains more than one *Solution Snippet* to solve the question.
The code block that contains the *Solution Snippet* also contains sample inputs/outputs	The code block contains an exemplary usage of the *Solution Snippet* (i.e., inputs and/or outputs) in addition to the *Solution Snippet*.
The code blocks in the answer contain — in addition to the *Solution Snippets*— inputs and code snippets copied from the question's body	The *Solution Snippet* in the code block contain code snippets, inputs, and/or data provided in the question's body.
The *Solution Snippet* is an example of a programming concept	The *Solution Snippet* is not a precise solution, and it is an example of a programming concept.

The answer does not contain *Solution Snippets*	No *Solution Snippet* is found in the answer.
The answer contains *Solution Snippets* for related problems	The answer contains at least one *Solution Snippet* that is not a solution to the original question being asked in the title of the question, but it is a *Solution Snippet* to a closely related question.

1.5.2 A categorization of the reusability aspects of solution snippets

This is the answer to RQ2. It has five categories, and they are briefly described in Table 1.2.

Table 1.2: Categories for the reusability of solution snippets (the answer to RQ2).

Categories	**Description**
As-is	The *Solution Snippet* does not require any changes to be adapted to the target source code, and it could be reused as is with a simple copy-and-paste
User-defined functions	The *Solution Snippet* is presented as an entire function (a user-defined function). Thus, it is ready to be reused but requires a function call in order to be adapted to the target source code.
Placeholders	The *Solution Snippet* contains sections that need to be completely removed or replaced in order to be adapted to the target source code. Such sections are called placeholders in this thesis.
Customizable variables and/or data	The *Solution Snippet* contains variables and/or data that require customization in order to be adapted to the target source code.
Conceptual	The *Solution Snippet* exemplifies a certain concept in the programming language. Thus, such *Solution Snippets* are often not straightforward to be adapted to the target source code. It is a concept that should be understood and adapted for reuse.

1.5.3 An analysis of the above two categorizations.

By reflecting on the first categorization (how solution snippets are presented in answers), I identified **a list of potential reasons why it is challenging to find and extract *Solution Snippets***:

- *Solution Snippets* may be found either within code blocks or inline with the text. Notably, some answers contain more than one *Solution Snippet* (either within the same or in separate code blocks) and *Solution Snippets* that solve a

different question other than the one in the question's title. Thus, it is not easy to identify where *Solution Snippets* are found in an answer to a Stack Overflow's question.

- A *Solution Snippet* is often surrounded by ancillary code (source code that is not part of the actual *Solution Snippet* to the problem) within a code block, and it is challenging to separate a *Solution Snippet* from ancillary code. There are several types of ancillary code: (1) example uses of the *Solution Snippet*, (2) the output of the *Solution Snippet* and (3) supplementary code snippets such as variable initializations.

In the second categorization (the reusability aspects of solution snippets), the first two categories (As-is, User-defined functions) appear easy to be reused. As-is *Solution Snippets* can be copied and pasted when reusing, and User-defined functions require a function call, often specifying the function arguments. The last three categories (Placeholders, Customizable variables and/or data, Conceptual) appear challenging to reuse and require understanding the source code to be customized:

- There are several types of placeholders (e.g. print statements, comments, call to `alert` function, etc.), and there is no consistency of authors on what to use as placeholders; for example, an author may use a print statement, while another author uses an ellipse as a placeholder. When adapting *Solution Snippets* containing placeholders, the placeholders need to be identified and replaced with the actual source code.

- Customizable variables and/or data must be identified and replaced with the actual variables and values in the target source code where the *Solution Snippet* is reused.

- Conceptual *Solution Snippets* appear to be the most challenging ones to be reused. How such *Solution Snippets* need to be adapted for reuse is not apparent.

1.5.4 A manually annotated dataset

For reproducibility and traceability of this research, the manually annotated dataset is made available at [6]. This dataset consists of 300 *how-to* questions and their answers that were coded during this study. More details about this dataset are provided in Appendix C.

1.6 Thesis Outline

This thesis is organized as follows.

Chapter 2 elaborates the background information on the motivation for this work, including the limitations of the existing datasets of Question-Solution pairs and the existing categorizations that describe the nature of questions and answers on Stack Overflow.

Chapter 3 presents this study's goal and research questions, the philosophical worldview taken to design this study, the research design, and my role as a researcher in this study.

Chapter 4 presents the findings of this study, the two categorizations: (1) how *Solution Snippets* are presented in answers and (2) how *Solution Snippets* need to be reused.

Chapter 6 describes how the results of this study further expand and complement the prior work.

Chapter 5 discusses the findings of this study and the threats to the validity.

Chapter 7 concludes the study and describes the future directions.

2

Background

I begin this chapter with a description of the nomenclature used by other researchers regarding *Solution Snippets* in Stack Overflow and the definition of a *Solution Snippet* within the context of this thesis (Section 2.1). Then, I introduce the existing datasets of Question-Solution pairs and describe their limitations (Sections 2.2 and 2.3). Finally, I present the prior categorizations related to Stack Overflow's *Solution Snippets* and explain why new categorizations are needed to understand the potential reasons for the limitations of the existing datasets (Section 2.4).

2.1 *Solution Snippets*

Even though there are several methods to create datasets of Question-Solution pairs, there is no agreed definition of what a *Solution Snippet* of Stack Overflow's question is. Allamanis et al. [4] and Iyer et al. [28] considered a code block as a *Solution Snippet*. Yao et al. [53] defined a *code solution* as a code block that contains a solution to a specific programming problem. Yin et al. [54] referred to the contiguous lines[1] of source code within a code block "that implements the description in a question's title" as a *snippet*.

The existing definitions have a Stack Overflow-centric view of *Solution Snippets* within code blocks. In the context of this study, I want to understand what is a *Solution Snippet* beyond code blocks in Stack Overflow. Thus, I define a *Solution Snippet* as one or more portions of source code that solve a specific programming question (a portion could be a part of a line or several lines). This definition is independent of

[1]For example, the contiguous lines in code block having three lines are line 1, line 2, line 3, lines 1&2, lines 2&3, and lines 1,2,&3.

code blocks in Stack Overflow, unlike the existing definitions; a *Solution Snippet* could be inline with the text or within code blocks; a *Solution Snippet* within a code block may not span over the entire code block and could be composed of either contiguous or non-contiguous lines of source code; a *Solution Snippet* may span over multiple code blocks as well. For example, Figure 2.1 shows an excerpt from the accepted answer to the question *"How do I sort a dictionary by value?"* (only two of the six code blocks that belonged to this answer are shown). Each of these two code blocks contains a *Solution Snippet*, which is annotated in blue. Each *Solution Snippet* in this example does not span the entire code block and is surrounded by ancillary code (the source code that is not part of the *Solution Snippet* within a code block is called ancillary code): the REPL[2] prompt (>>>), the code required to demonstrate the use of the *Solution Snippet*, and the output of the REPL. To create a Question-Solution pair, an automatic extractor needs to extract any of these *Solution Snippets* and pair them with the question's title.

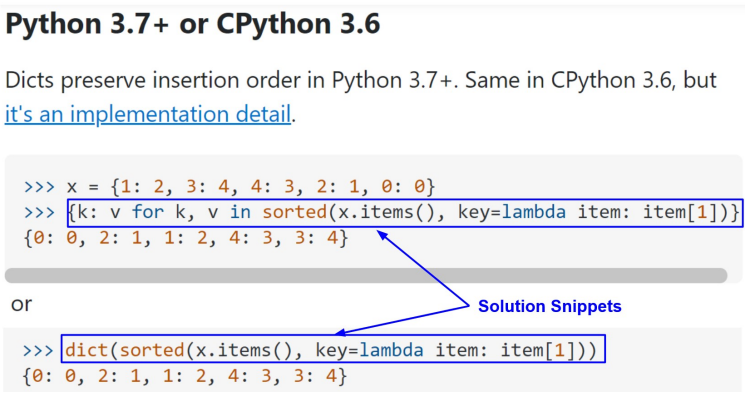


Figure 2.1: An excerpt of an accepted answer showing its first two code blocks. Each of them contains a *Solution Snippet* that is surrounded by ancillary code.

In this case, the ancillary code within the first code block contains key-value pairs that are not relevant to the question and not part of the *Solution Snippets*. However, this ancillary code shows how the variable `x` needs to be initialized for reuse. Because such ancillary code contains contextual information about the reusability of *Solution Snippets*, it is potentially useful when training machine learning models to automatically generate source code well adapted to a target source code. Thus, it would be useful to extract the *Solution Snippet* as well as such ancillary code within code blocks when creating datasets for machine learning purposes.

[2]Read-eval-print loop or language shell

In some cases, neither the *Solution Snippet* nor the ancillary code contain contextual information about how a *Solution Snippet* need to be adapted for reuse. In such cases, it would be useful to tag different parts in a *Solution Snippet* that need to be adapted for reuse. For example, Figure 2.2 shows an excerpt of the accepted answer to the question *"How do you clone an array of objects in JavaScript?"* Its code block contains a *Solution Snippet* (annotated in blue) that does not span the entire code block; the variable that holds the result of this *Solution Snippet* is ancillary code which is not part of the *Solution Snippet*. In this case, neither the *Solution Snippet* nor the ancillary code show that variable `nodesArray` in the *Solution Snippet* needs to be replaced with the variable name of an array (and/or initialized with an array) when reusing the *Solution Snippet*.

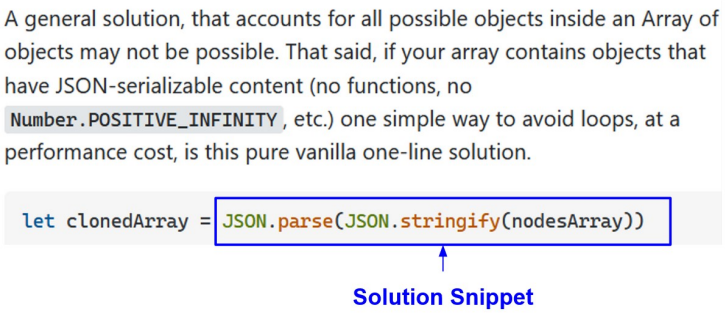


Figure 2.2: An excerpt of an answer having a *Solution Snippet* with a variable named `nodesArray` that needs to be replaced when reusing.

2.2 Datasets of Question-Solution pairs that were extracted from Stack Overflow and their limitations

Researchers have been extracting *Solution Snippets* from Stack Overflow to create datasets of Question-Solution pairs:

- Yan et al. [51] manually annotated 26 Question-Solution pairs by selectively choosing posts from Stack Overflow by manually checking the correctness of *Solution Snippets* and correcting when required.
- Allamanis et al. [4] created a dataset of Question-Solution pairs from Stack Overflow by using all the code blocks in its answer as *Solution Snippets*. How-

ever, they ignored the code blocks that contain more than 300 characters, assuming that shorter code blocks will be more likely to contain *Solution Snippets*.

- Iyer et al. [28] created a dataset of Question-Solution pairs by pairing the titles of *how-to* questions with the code blocks in their accepted answers if those answers contain only one code block. They discarded all the questions that contain multiple code blocks in the accepted answer (thus ignoring many potential Question-Solution pairs).

- Yao et al. [53] trained a neural network to extract a dataset of Question-Solution pairs from Stack Overflow: given a *how-to* question and its accepted answer as input, this neural network predicts whether each code block in this answer is a *Solution Snippet* to the question or not. They paired the code blocks predicted as *Solution Snippets* with the question's title to create Question-Solution pairs.

- Yin et al. [54] created a dataset of Question-Solution pairs from Stack Overflow using a machine learning model. Given a code block in an answer to a *how-to* question as input, their machine learning model is capable of classifying contiguous lines of source code within the code block into two classes: *"a snippet"* or not. They paired the title of a question with the contiguous lines of source code that were classified into the class *"a snippet."* This model was initially trained using a large dataset of Question-Solution pairs extracted by pairing the title of a question to the only code block in its accepted answer (as done in [28]). Later, the model was fine-tuned using 254 manually extracted Question-Solution pairs.

Table 2.1 shows the limitations of these datasets. While Stack Overflow provides a method to specify code blocks using its Markdown dialect, it does not help distinguish between code blocks that contain only *Solution Snippets* and ones that contain ancillary code. Thus, a major limitation of several of these datasets ([4,28,53]) is that they consider a *Solution Snippet* to be the entire code block (thus including ancillary code). To address this issue, some researchers have ignored large code blocks and answers that contain more than one code block [4, 28]. In addition, the authors of two datasets that were created using machine learning have acknowledged that some extracted pairs contain incorrect *Solution Snippets* [53, 54]. Other researchers also have shown that Question-Solution pairs extracted from Stack Overflow are of poor quality, and they have discussed the need for a better dataset of pairs [31, 50].

Table 2.1: Limitations of existing datasets of Question-Solution pairs created from Stack Overflow

Source	**Description of the dataset**	**Limitations**
Yan et al. [51]	This dataset contains 26 Java Question-Solution pairs that were manually curated.	• According to [50], manually curated datasets may be too small to train machine learning models.
Iyer et al. [28]	This dataset contains 145,841 Question-Solution pairs of C# and 41,340 Question-Solution pairs of SQL. A pair was created by pairing the question's title to the code block in its accepted answer.	• The authors discarded all the accepted answers with more than one code block. • They considered that a *Solution Snippet* spans over an entire code block. • Another study has stated that this dataset contains pairs of poor quality [31].
Allamanis et al. [4]	This dataset contains 53,743 Question-Solution pairs of C#. A pair was created by pairing the question's title to all the code blocks in all of its answers.	• The authors considered that a *Solution Snippet* spans over an entire code block. • They removed the code blocks that contain more than 300 characters, assuming that only shorter code blocks will contain *Solution Snippets*.
Yao et al. [53]	This dataset contains 148k Question-Solution pairs in Python and 120k Question-Solution pairs in SQL that were automatically mined using machine learning.	• According to its authors and others [31], this dataset contains Question-Solution pairs with incorrect *Solution Snippets*. • The model used to mine this dataset considered that a *Solution Snippet* spans over an entire code block.

Yin et al. [54]	This dataset contains 600k Question-Solution pairs of Java/python automatically mined using machine learning.	This dataset contains low-quality pairs [50]. The authors acknowledged that the machine learning model used to create this dataset: • is biased towards detecting a large number of contiguous lines of code as a *Solution Snippet* to a question, • detects incorrect *Solution Snippets* to some questions, • identifies examples as *Solution Snippets* to some questions, and • identifies *Solution Snippets* and the ancillary code snippets that surrounds the *Solution Snippets* (such as examples, REPL outputs, etc) as *Solution Snippets* to some questions. Lastly, *Solution Snippets* composed of non-contiguous snippets within a code block are not identified by the machine learning model used to create this dataset.

In contrast to automatically curated datasets, manually curated datasets usually contain high-quality pairs. However, there are several limitations; on the one hand, manually curated datasets often contain a small number of pairs and are not sufficient to train machine learning models [31]; on the other hand, Stack Overflow is not optimized to highlight the *Solution Snippet* (sometimes there are multiple *Solution Snippets*), and the researcher has to read the whole answer to identify the *Solution Snippet*, which is labour-intensive.

Lastly, for machine learning applications to generate *Solution Snippets* adapted to the target source code, the information about the possible ways of adapting a *Solution Snippet* in the training dataset is potentially useful. Such information is not considered when creating the existing datasets.

2.3 Datasets of Question-Solution pairs that were extracted from non-Stack Overflow sources and their limitations

Even though the context of this thesis is Stack Overflow, other sources (such as GitHub[3], API documentations, etc.) have been used to create datasets of pairs. Researchers who used GitHub to create datasets have paired docstrings of Python functions to their function bodies — a language-specific method [9, 16, 27, 29]. However, such datasets often contain poor-quality Question-Solution pairs [27]. Besides, not every project on GitHub has docstrings (that describe the source code) declared, and extracting the natural language text from GitHub may not always be straightforward [9]. In Oda et al.'s [37] study, human annotators manually curated a dataset of Question-Solution pairs from GitHub even though it was a labour-intensive task. Lu et al. [32] created a dataset of Question-Solution pairs using questions in the web query logs of a commercial search engine. For each programming-related question in the logs, the authors paired the top two *Solution Snippets* in Husain et al.'s dataset ([27]) that shows the highest *similarity*[4] scores (to the question) computed by an existing code search model ([20]). In addition, researchers have mined tutorials and

[3] `https://github.com/`

[4] Feng et al.'s [20] model (CodeBERT) first encoded the questions and the *Solution Snippets* in a dataset as vectors (in a vector space). Then, the model calculated the *similarity* score between a given question and a *Solution Snippet* by taking their dot product. The top *Solution Snippet* that matches a given question is the *Solution Snippet* that has the highest dot product with the question.

API documentations to create datasets [14, 50]. According to other studies, the *Solution Snippets* found in such sources are simpler than those required in practice in order to train better machine learning models [9, 48]. Lastly, researchers have also created datasets that are limited to specific domains, such as the dataset of Text-Bash command pairs created by Lin et al. [31].

2.4 Categorizations of Stack Overflow's questions and answers, and their limitations

Previous researchers have categorized Stack Overflow's questions and answers, and how developers reuse Stack Overflow's code blocks. The following are the limitations of the existing categorizations with respect to locating, extracting and reusing *Solution Snippets* (see Table 2.2).

- Allamanis et al. [3], Rosen et al. [44], Treude et al. [46], and Beyer et al. [10–12] presented categorizations of Stack Overflow's questions, but they did not inspect the *Solution Snippets* in answers.
- Nasehi et al. [34] categorized Stack Overflow's questions and the recognized answers[5] to those questions. They did not consider whether a code block contains a *Solution Snippet* or not.
- Zagalsky et al. [55] categorized questions and answers posted on Stack Overflow and the R-help mailing list. When categorizing answers, they focused on the text, ignoring source code; their categories describe how the text in answers explains the code blocks (for example, whether the text describes the code blocks step by step).
- Wu et al. [49] categorized how Stack Overflow's code blocks attributed in GitHub; their categories describe how developers had adapted the source code that originated from Stack Overflow in GitHub projects.

[5] In [34], a recognized answer is an answer that is either an accepted answer or an unaccepted answer with a normalized score of 0.4 or more.

In summary, there are several limitations of the existing categorizations with respect to how *Solution Snippets* are located, extracted and adapted for reuse. First, some categorizations only described how questions look like but not answers [3, 10–12, 44, 46]. Second, the categorizations that described how answers look like did not differentiate between code blocks that contain *Solution Snippets* and the ones that do not [34, 49, 55]. Third, other categorizations barely described how *Solution Snippets* need to be adapted for reuse [34, 55]. Lastly, the categorizations that described the reusability of *Solution Snippets* ([49]) looked at how developers actually reuse Stack Overflow's *Solution Snippets* on GitHub, but not the potential ways to adapt *Solution Snippets* (which is the information needed for machine learning models).

Table 2.2: Existing categorizations and limitations with respect to locating *Solution Snippets* in answers and the reusability of *Solution Snippets*

Source	**Description of the research**	**Limitation**
[3, 10–12, 44, 46]	• In all these studies, the authors categorized questions in Stack Overflow.	• Inspected the questions in Stack Overflow and not the answers.
Nasehi et al. [34]	• Categorize 163 Java Stack Overflow questions that contain code blocks in their answers. • Identified four categories of questions and nine categories (which they called attributes) of recognized answers in Stack Overflow.	• Did not consider whether code blocks contain a *Solution Snippet* or not.
Zagalsky et al. [55]	• Categorized *R* questions posted on Stack Overflow, their answers, and emails in the R-help mailing list. • Identified ten categories of questions and nine categories of answers in Stack Overflow. • Described how the text in answers explains the code blocks: for example, whether the text is like a tutorial that explains the code step by step, whether the text provides clues or suggestions without actually solving the question and whether the text provides an external link to a solution, etc.	• Did not describe the source code within code blocks in answers: for example, whether the code block contains a *Solution Snippet*, whether the code block contains ancillary code, etc.
Wu et al. [49]	• Studied source code that originated from Stack Overflow and attributed in GitHub projects, concentrating on how they had been adjusted for reuse in GitHub.	• Categorized how source code originated from Stack Overflow is utilized by developers in GitHub (irrespective of the potential aspects of reusability of the Stack Overflow's source code).

3

The Approach

Researchers require high-quality datasets of Question-Solution pairs to train better machine learning models. However, there are limitations in existing datasets [31, 50] (as described in Chapter 2); on the one hand, there is a trade-off between the quality of pairs and the number of pairs when creating a dataset; on the other hand, it is not easy to identify how *Solution Snippets* need to be adapted for reuse and incorporate that information into datasets. This chapter aims to describe an approach to understand why the aforementioned are challenging. I begin this chapter by presenting the goal and research questions that guided this study (Section 3.1). Then, I describe why the *constructivist* worldview is chosen to design this study, and I detail the research design (Sections 3.2 and 3.3). I end this chapter by describing my role as a researcher in this study (Section 3.4).

3.1 Research Goal and Research Questions

The basic conjecture of this thesis is that *the understanding of how Solution Snippets are presented in Stack Overflow's answers and how Solution Snippets need to be reused* will help researchers improve current methods to create datasets of pairs from Stack Overflow. Hence, the overarching goal of this study is to:

> "Understand how *Solution Snippets* are presented in Stack Overflow's answers and how those *Solution Snippets* need to be adapted for reuse."

The two research questions addressed in this study are:

- *RQ1. How are* Solution Snippets *presented in Stack Overflow's answers?*
- *RQ2. How are* Solution Snippets *potentially adapted in order to be reused?*

3.2 The Philosophical Worldview of this Study

According to Creswell et al. [17], identifying the philosophical worldview will help explain why a researcher chose a qualitative, quantitative, or mixed-methods approach for the research. I begin describing what a *philosophical worldview* is and the four most prominent philosophical worldviews. Then, I describe why this study takes a *constructivist* worldview to design its methodology.

In literature, there are several definitions for the *philosophical worldview*. Guba et al. [23] define the *philosophical worldview* as *"a basic set of beliefs that guide actions."* Cresswell et al. [17] define the *philosophical worldview* as *"a general philosophical orientation about the world and the nature of research that a researcher brings to a study."* Other nomenclatures used for the *philosophical worldview* are *epistemologies and ontologies*, and *paradigms* [19,24]. There are four prominent philosophical worldviews:

Postpositivist worldview (reductionist method). The research focuses on *"identifying and assessing the causes that influence outcomes, such as those found in experiments"* [17]. The researchers tend to leverage quantitative research more than qualitative research. The researchers reduce the ideas into a collection of "variables that comprise hypotheses and research questions." Then, the researchers collect data and check if the data support the hypotheses or not [17].

Constructivist worldview. The research focuses on *"understanding the world or making sense of the meanings that others have about the world"* [17]. The research problems are broadly defined, and research questions are often exploratory. The researchers tend to leverage qualitative research, and those methods are usually emergent.

Transformative worldview. The research *focuses on the marginalized groups* in society and their needs [17]. This type of study also leverages qualitative methods. Unlike the constructivist worldview, in the transformative worldview, researchers may get help from the participants of the study to collect data, analyze data, etc. By doing so, the researchers raise awareness among the marginalized people and may propel a plan to improve their lives [17].

Pragmatic worldview. The researchers emphasize the research problem and leverage all the suitable approaches that best meet the requirements of the research problem [17]. Therefore, pragmatists often use mixed methods that are comprised of both qualitative and quantitative methods [17].

The research questions of this study are exploratory because they aim to explore how *Solution Snippets* are presented in Stack Overflow's answers and how *Solution Snippets* are potentially adapted for reuse. Therefore, this study takes the ***constructivist worldview***.

As mentioned before, the *constructivist worldview* is usually an approach to ***qualitative research*** [17]. In contrast to quantitative methods that measure variables and test hypotheses, qualitative methods are designed to organize data into codes to build patterns and categories — and in some cases, induce theories [18, 25]. Accordingly, I designed a qualitative study to find answers to the research questions by coding Stack Overflow's *Solution Snippets*.

3.3 Research Design

The qualitative study designed in this research is illustrated in Figure 3.1. It has two main parts:

1. **Identifying a collection of *how-to* questions which are likely to contain *Solution Snippets* in their answers:** In order to study *Solution Snippets*, I needed a collection of *how-to* questions. Accordingly, 300 top-scored *how-to* questions were identified from the SOTorrent database [8] (Sections 3.3.1 and 3.3.2).

2. **Coding the answers to these *how-to* questions**: By coding the top-scored answers to *how-to* questions, two categorizations were identified (Section 3.3.3). In particular, for RQ1, a categorization of how *Solution Snippets* are presented in answers was identified. For RQ2, a categorization of how *Solution Snippets* need to be adapted for reuse was identified.

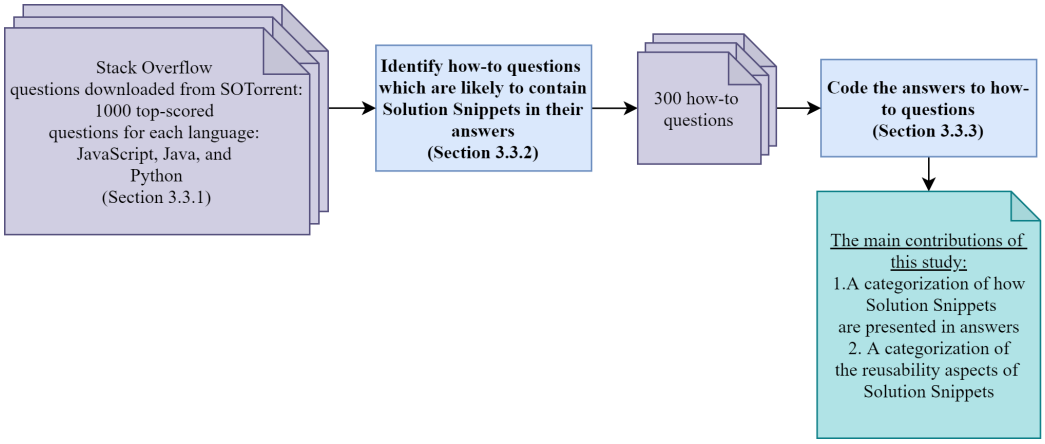


Figure 3.1: An overview of the research design

The design is described in detail below.

3.3.1 Data Collection

For this research, I chose the three most popular[1] programming languages on Stack Overflow: *JavaScript, Java,* and *Python.* For each language, I obtained the 1000 top-scored questions from the SOTorrent database [8] (the schema is available in appendix A). Since only *how-to* questions are likely to contain *Solution Snippets* in their answers, I had to identify a collection of *how-to* questions to study *Solution Snippets*.

3.3.2 Identifying *how-to* questions which are likely to contain *Solution Snippets* in their answers

A *how-to* question in Stack Overflow is a question that is likely to be answered with *Solution Snippets*. The process to distinguish *how-to* questions from other questions required an analysis of the questions obtained from the SOTorrent database. The

[1]https://stackoverflow.com/tags

process was iterative, as shown in Figure 3.2. In each iteration, three coders (including myself) classified a batch of Stack Overflow's questions into two main classes: "a *how-to* question" or "not a *how-to* question." The coders iteratively classified batches of questions until the *Fleiss' Kappa Statistic* had exceeded a threshold of 0.9, which is higher than the substantial strength of agreement suggested by Landis et al. [30] (the *Cohen's Kappa coefficient* is not used since there were more than two coders).

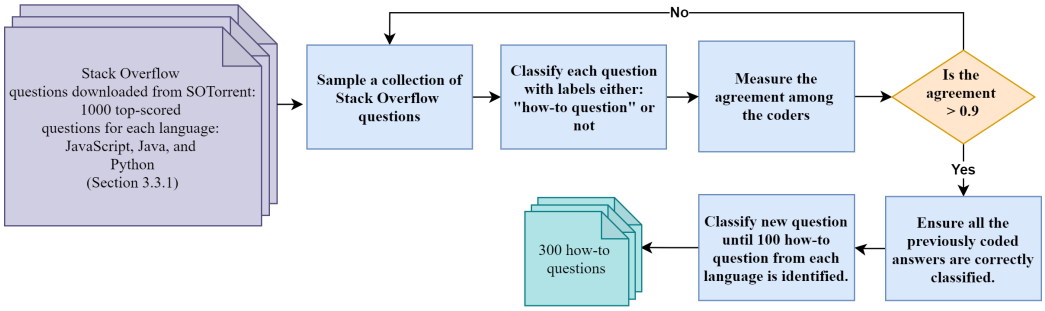


Figure 3.2: The process I used to identify *how-to* questions.

I describe the process followed in a single iteration below.

1. **Sampling a batch of Stack Overflow questions:** The score of the question was used as a proxy for importance when selecting questions, and in each iteration, the next 15-60 top-scored questions were obtained. To classify these questions, I prepared three identical Google sheets, one for each coder. Each Google Sheet included three main columns: *"question's title,"* which was populated with the titles of the questions selected for the iteration, *"question's URL,"* which was populated with the URLs of those questions, and *"class,"* which was left empty for the coders to indicate the class of each question. I distributed the editable links to these Google sheets among the three coders to classify questions independently.

2. **Classifying each question either as a *how-to* question or not:** Because the title is often sufficient to classify a question, each coder read the titles of the questions (provided in the Google sheet for the particular iteration) to classify them; according to the title, if a question was likely to contain *Solution Snippets* in its answers, the coder classified it as *"a how-to question,"* and *"not a how-to question"* otherwise. Especially in the first three iterations, if the class of a question was unclear, the coder navigated to the URL and looked at the body of the question to classify it; if the class was further unclear, the coder labelled

it *"maybe a how-to question."* Having the latter class allowed some freedom for the coders to choose neither of the main classes: *"a how-to question"* nor *"not a how-to question."*

In the subsequent iterations, choosing between two classes instead of three forced the coders to select either of the two main classes. By doing so, it ensured high agreement levels among the coders.

3. **Calculating the intercoder agreement, discussing the disagreements, and revising the classification:** At the end of each iteration, I measured the level of agreement among the coders by calculating the *Fleiss' Kappa Coefficient*. When there was a disagreement on a particular question's class (including the ones that were classified as *"maybe a how-to question"* by at least one coder), the coders discussed it and refined the understanding of the classification, and then, the coders revised the classes of such questions.

Table 3.1: The number of questions classified, the number of *how-to* questions, the number of coders, the agreement at the end of iteration

Iteration	**The number of questions classified**	**Number of *how-to* questions identified**	**The number of coders**	**The Fleiss' kappa statistic**
1	21	12	3	0.22
2	21	7	3	0.41
3	24	17	3	0.45
4	63	43	3	0.82
5	18	14	3	0.71
6	15	13	3	0.96
7	278	194	1	NA
	440	300		

The agreement in the first three iterations was low (below the substantial agreement suggested by Landis et al. [30]), but it improved over the subsequent iterations. At the end of the sixth iteration, the Fleiss' Kappa statistic was 0.96 with a p-value of zero, which showed almost perfect agreement. By then, 106 out of 162 (65.4%) questions had been identified as *how-to* questions in all iterations. I re-classified those 162 questions to ensure the classification was consistent across previous iterations. Then, I continued classifying more questions until the number of *how-to* questions for each language reached 100. In total, 300 out of 440 (68.2%) questions were identified as

how-to questions. The sample of 440 questions comprised 135 JavaScript questions, 157 Java questions, and 148 Python questions. Table 3.1 presents the number of questions classified in each iteration, the number of *how-to* questions identified, the number of coders, and the level of agreement among the coders.

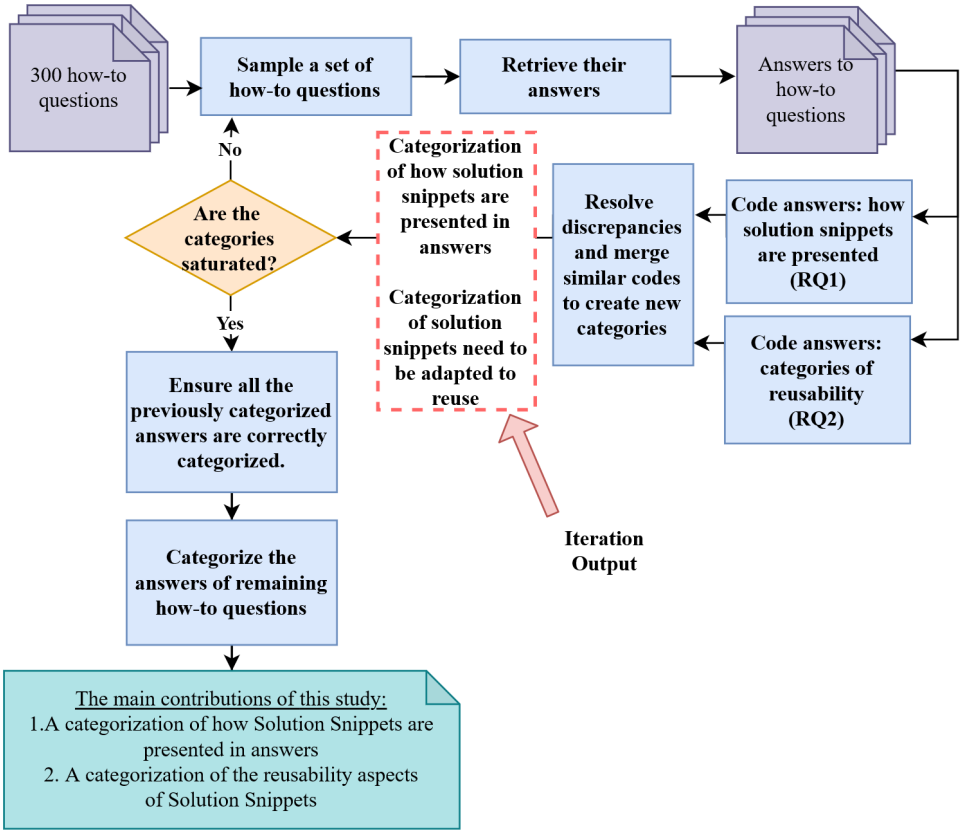


Figure 3.3: The methodology used to study Stack Overflow's answers and *solution snippets*.

3.3.3 Coding the answers to *how-to* questions

Once a collection of *how-to* questions was identified, the three coders (including myself) coded the answers to those questions in order to answer the two research questions of this study. Since the coders started coding without any preconceived ideas of how the answers and the *Solution Snippets* in those answers looked like, the process was iterative, as shown in Figure 3.3. In a qualitative study like this, the categories are assumed to be saturated when previously unseen (new) data no longer reveals new categories [15, 17, 26]. Thus, the coders coded batches of answers (132 answers) until the saturation of categories was reached. Lastly, I continued coding (466) answers

to the remaining *how-to* questions, ensuring that premature closure of coding [2] was avoided. This coding produced a categorization for each research question.

I describe the process followed in a single iteration below.

1. **Sampling a batch of *how-to* questions:** In each iteration, the next 15-60 top-scored *how-to* questions were used. In the first iteration, only the accepted answers to the *how-to* questions (used for the iteration) were analyzed. At this point, it appeared that some questions contained competing answers having scores higher than the accepted answer. In subsequent iterations, the coders analyzed the two top-scored answers to each question irrespective of being an accepted answer or not. Because the top-scored answers are more likely to contain the most useful *Solution Snippets* to pair with titles to create Question-Solution pairs, studying only the top-scored answers per question is not a critical threat to the validity of the results.

2. **Coding the answers and solution snippets, and resolving discrepancies between coders:** In each iteration, I prepared three identical Google sheets (one for each coder) to code the answers. The columns were: *"the how-to question's title," "the answer's URL,"* a group of columns for categories (one column for each category identified at the end of the previous iteration) for corresponding research questions, and a *"comment"* column for each research question. Then, I distributed the editable links to these Google sheets among the coders. The coders were supposed to navigate to the answers' URLs, study the *Solution Snippets* in them, and independently code them in the Google Sheets.

   In the first iteration, the coders did not have any predefined categories for RQ1 and RQ2, and they independently suggested categories for answers in the *"comment"* columns of the two research questions. Then, the coders discussed the suggested categories and merged the similar categories. After merging the similar categories, the categories for RQ1 at the end of the first iteration were: "*Solution Snippet is a one-liner*" (e.g. one function call) and "*Solution Snippet contains multiple function calls.*" The categories for RQ2 were: "*Solution Snip-*

[2] Premature closure of coding occurs when the categories remain unchanged for quite a while, and the coders stop coding, assuming the categories are saturated. However, the sample of data used for coding is not adequate, and coding new data likely reveal new categories.

pet is a function definition" (a generic *Solution Snippet*), and "*Solution Snippet contains specific data & variables*" (a specific *Solution Snippet*).

In following iterations, the coders independently categorized answers into the categories that were identified at the end of the previous iteration; for each answer, the coders checked (✓) the columns of the categories that the *Solution Snippets* in that answer belonged to. If *Solution Snippets* in an answer did not belong to any category in the Google sheet, the coder suggested a new category in the corresponding research question's "comment" column. At the end of each coding iteration, the coders discussed their discrepancies (in assigning categories) and resolved those discrepancies. In addition, the coders merged any new categories (suggested by them) that looked similar, and occasionally, the coders merged the existing categories that looked similar.

By the end of the fifth iteration, the categories for the two research questions were saturated. By then, 132 answers across 102 *how-to* questions (across the three languages) had been categorized. In the fifth (final) iteration, I did a pass over the entire batch of 132 previously coded answers, thereby ensuring that each of them was assigned to one or more saturated categories. Then, I categorized the (466) answers to the remaining *how-to* questions. This was done to avoid premature closure of coding (as suggested in [2] and [5]) and to code more answers. By the end, 598 Stack Overflow answers across 300 *how-to* questions were categorized (see Table 3.2). Appendix B presents the number of answers studied in each iteration, the number of coders who participated in coding, and how the categories evolved over iterations.

Lastly, I wrote a description for each category by looking at the *Solution Snippets* in a random sample of answers that belonged to the category. Chapter 4 presents the two categorizations and a detailed description of each category.

Table 3.2: Total number of questions and answers studied

Language	**Number of *how-to* questions**	**Number of answers**
JavaScript	100	200
Java	100	198
Python	100	200
Total	300	598

3.4 Researcher's Role

The method chosen in this study is qualitative, and the researcher's role impacts the knowledge produced in a qualitative study [17]. The researcher's reflections on his/her role are necessary to understand the bias imposed on his/her interpretations [17]. In this study, three coders (including myself) classified questions and coded answers; I will first briefly describe who were the coders and what specific parts of the study they did. Then, I will describe my reflections on my role as a researcher to understand the possible bias I may impose on the interpretations in this thesis.

All the coders were programmers and had experience using Stack Overflow as users. All of them classified 162 questions (as mentioned in Section 3.3.2) until they reached an almost perfect agreement, and I classified the remaining 278 questions. In addition, the coders coded 132 answers (as mentioned in Section 3.3.3) until they reached the saturation of the categories, and I coded the remaining 466 answers to ensure that the coders avoided the premature closure of coding.

The bias imposed by a researcher varies depending on whether he/she is an *insider* or an *outsider* to the research [33]. In this study, I have been an *insider* as well as an *outsider*. According to Naaeke et al. [33], having both perspectives are *"consequential for research because they impact the research process, the findings of a study, and the argument made by the researcher about the implications of these findings."* I am an insider because I have been using Stack Overflow to seek solutions for programming problems for over five years, and I understand how Stack Overflow works and how a typical user uses it. I am an outsider because I am not the author of any of Stack Overflow's questions and answers studied in this research, and I believe that helped me code Stack Overflow's questions and answers impartially.

3.5 Summary

To understand how *Solution Snippets* are presented in Stack Overflow's answers and how those *Solution Snippets* need to be adapted for reuse, this study took a *constructivist worldview* — an approach to *qualitative research*. Accordingly, I designed a qualitative study; three coders (including myself) identified *how-to* questions on Stack Overflow and categorized their answers. This resulted in two categorizations, each answering my two research questions. I believe having both *insider* and *outsider* perspectives helped me categorize Stack Overflow's answers impartially.

4

Findings of the Study

In this study, two categorizations were produced to answer my two research questions, and they are described in detail in Section 4.1 and Section 4.2.

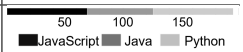
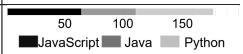
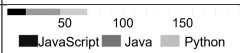
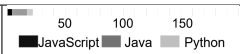
4.1 RQ1. How are solution snippets presented in Stack Overflow's answers?

For RQ1, this study produced a categorization of how *Solution Snippets* are presented in Stack Overflow's answers. There are three main categories: (1) *Solution Snippets* inline with the text, (2) *Solution Snippets* within code blocks, and (3) answers that do not contain *Solution Snippets* to the original question, and/or the answers that contain *Solution Snippets* to related questions (instead of the original question). The above categories contain eight sub-categories, as shown in Table 4.1, and in this section, I describe them in detail.

4.1.1 The solution snippet is inlined with the text.

This category corresponds to any *Solution Snippet* that is embedded within the text in an answer. In this thesis, such *Solution Snippets* are called *inline Solution Snippets*. For example, in the question *"How to re-import an updated package while in Python Interpreter?"*, the second top-scored answer (shown in Figure 4.1) provided two *Solution Snippets* inlined with the text (annotated in red). Of the answers that were studied in this research, 9.5% contained inlined *Solution Snippets*.

Table 4.1: The answer to RQ1. How are *Solution Snippets* presented in Stack Overflow's answers?

RQ1. How are solution snippets presented in Stack Overflow's answers?	**Frequency of answers**	**Distribution of frequencies**
***Solution Snippets* inline with text**		
1. The *Solution Snippet* is inlined with the text: the *Solution Snippet* appears inline with *natural language* text in the answer.	58	
***Solution Snippets* within code blocks**		
2. The code block contains only the *Solution Snippet*: the answer contains only one code block and, the *Solution Snippet* spans this entire code block.	163	
3. The answer contains more than one *Solution Snippet*: the answer contains more than one *Solution Snippet* to solve the question.	195	
4. The code block that contains the *Solution Snippet* also contains sample inputs/outputs: at least one code block in the answer contains an exemplary usage of the *Solution Snippet* (i.e., inputs and/or outputs) in addition to the *Solution Snippet*.	178	
5. The code blocks in the answer contain — in addition to the *Solution Snippets*— inputs and code snippets copied from the question's body: the *Solution Snippet* within a code block contains code snippets, inputs, and/or data provided in the question's body.	189	
6. The *Solution Snippet* is an example of a programming concept: the *Solution Snippet* is not a precise solution, and it is an example.	69	
Answers that do not contain *Solution Snippets* to the original question, and/or answers that contain *Solution Snippets* to related questions instead of the original question.		
7. The answer does not contain *Solution Snippets*: either a solution is described in the text without a precise *Solution Snippet*, or no *Solution Snippet* is found in the answer.	22	
8. The answer contains *Solution Snippets* for related problems: the answer contains at least one *Solution Snippet* that is not a solution to the original question being asked in the title of the question, but it is a solution to a closely related question.	41	

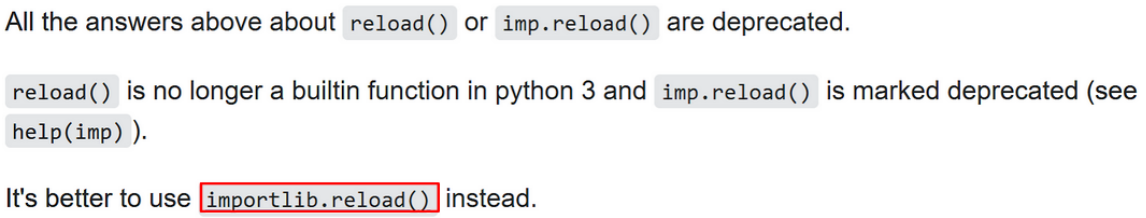


Figure 4.1: An answer with an inline *solution snippet* (annotated in red).

4.1.2 The code block contains only the solution snippet.

This category corresponds to any *Solution Snippet* that spans the only code block in an answer. Figure 4.2 shows the second top-scored answer to the question *"Is object empty?"* This answer contains only one code block, and it contains a *Solution Snippet* that spans the entire code block: `Object.keys(obj).length === 0`.

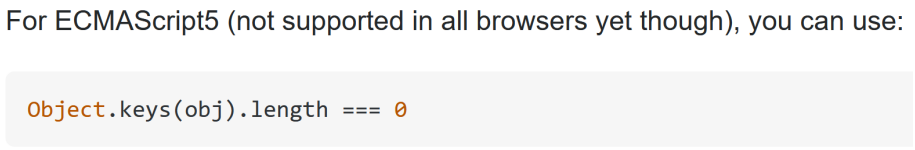
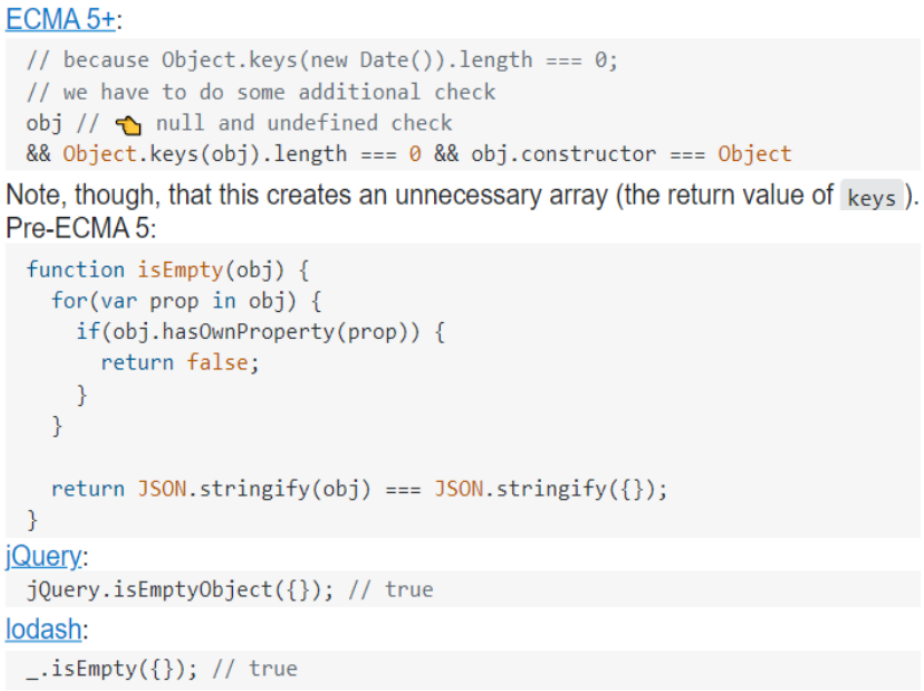


Figure 4.2: An answer containing only one code block in which the *solution snippet* spans the entire code block.

4.1.3 The answer contains more than one solution snippet.

This category corresponds to any answer that contains more than one *Solution Snippet*; in the sample of answers studied in this research, 34.5% of the JavaScript answers, 28.5% of the Java answers, and 34.8% of the Python answers contained multiple *Solution Snippets*. There are several reasons why this happens. First, **various versions of the programming language might require different *Solution Snippets***. Such cases were frequently observed in answers to JavaScript and Python questions (there were 35 JavaScript answers, 31 Python answers, and 16 Java answers that contained *Solution Snippets* applicable to different programming language versions). Second, **various libraries for the programming language might offer different *Solution Snippets*** (22 of the answers I studied contained such *Solution Snippets*). For example, the first four out of nine *Solution Snippets* in the accepted answer to the question *"How do I test for an empty JavaScript object?"* are shown in Figure 4.3. The first two *Solution Snippets* correspond to two different JavaScript versions, while

the other two *Solution Snippets* correspond to two different libraries. In some cases, multiple *Solution Snippets* may be contained in a single code block instead of separate code blocks (I will discuss these cases in detail in Section 6.2).


ECMA 5+:

```
// because Object.keys(new Date()).length === 0;
// we have to do some additional check
obj // null and undefined check
&& Object.keys(obj).length === 0 && obj.constructor === Object
```

Note, though, that this creates an unnecessary array (the return value of `keys`).
Pre-ECMA 5:

```
function isEmpty(obj) {
  for(var prop in obj) {
    if(obj.hasOwnProperty(prop)) {
      return false;
    }
  }

  return JSON.stringify(obj) === JSON.stringify({});
}
```

jQuery:

```
jQuery.isEmptyObject({}); // true
```

lodash:

```
_.isEmpty({}); // true
```

Figure 4.3: An answer with several blocks containing a *solution snippet*: the first two are specific to different JavaScript versions, and the next two use different libraries.

4.1.4 The code block that contains the solution snippet also contains sample inputs/outputs:

This category corresponds to any code block that contains an exemplary usage of the *Solution Snippet* in addition to the *Solution Snippet*. In particular, a *Solution Snippet* could be mixed with source code that corresponds to setting values of input variables (which are used by the *Solution Snippet*) and source code that displays the output variables' values. In some cases, the code block might also contain the output of the *Solution Snippet*, such as when the *Solution Snippet* is directly copied from the Python REPL. For example, Figure 4.4 shows a code block with the *Solution Snippet* in the accepted answer to the question *"How do I get python's pprint to return a string instead of printing?"*; the code snippets (a) and (c) together form the *Solution Snippet*, while code snippet (b) contains example data and code snippet (d) contains

the corresponding REPL output.

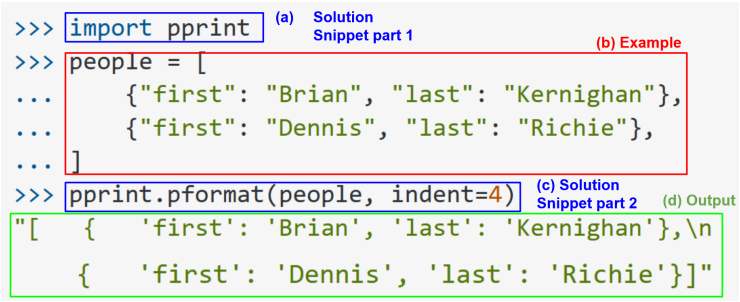


Figure 4.4: A code block with a *solution snippet*, its input, and its outputs.

4.1.5 The code block that contains the solution snippet also contains inputs and code snippets from the question's body.

This category corresponds to any *Solution Snippet* within a code block that contains source code, inputs, or data provided in the question's body. Whenever an answer contains *Solution Snippets* that are developed by either extending the source code or reusing data provided in the question's body, it might be useful to compare the *Solution Snippet* with the question's body in order to understand how the *Solution Snippet* needs to be adapted for reuse (this is further discussed in Section 6.3). For example, the question *"push multiple elements to array"* is shown in Figure 4.5, and the inline *Solution Snippets* in its accepted answer are shown in Figure 4.6. The *Solution Snippet* `a.push.apply(a,[1,2])` reuses the same variable (`a`) and values (`[1,2]`) as in the question. The question even presents a fragment of the *Solution Snippet*.

4.1.6 The solution snippet is an example of a programming concept.

This category corresponds to any *Solution Snippet* that is presented as an example that illustrates a programming concept. For example, in the question *"How to create a custom callback in JavaScript?"*, the entire code block in the accepted answer shown in Figure 4.7 is the *Solution Snippet*. Almost all the source code must be replaced when such a *Solution Snippet* is being reused. In particular, the functions `foo` and

I'm trying to push multiple elements as one array, but getting an error:

```
> a = []
[]
> a.push.apply(null, [1,2])
TypeError: Array.prototype.push called on null or undefined
```

I'm trying to do similar stuff that I'd do in ruby, I was thinking that `apply` is something like `*`.

```
>> a = []
=> []
>> a.push(*[1,2])
=> [1, 2]
```

Figure 4.5: The body of the question *"push multiple elements to array."*

In this case, you need `a.push.apply(a, [1,2])` (or more correctly `Array.prototype.push.apply(a, [1,2])`)

Figure 4.6: The accepted answer to the question in Figure 4.5 that reuses the same variable `a` and values `[1,2]`.

`do_something` represent two arbitrary functions in the target source code where this *Solution Snippet* needs to be reused, and the only reusable token appears to be the `callback`. Such a *Solution Snippet* exemplifies a concept rather than being a specific ready-to-use *Solution Snippet* (I discuss the reusability of *Solution Snippets* in detail in Section 4.2).

```
function doSomething(callback) {
    // ...
    // Call the callback
    callback('stuff', 'goes', 'here');
}
function foo(a, b, c) {
    // I'm the callback
    alert(a + " " + b + " " + c);
}
doSomething(foo);
```

Figure 4.7: A *solution snippet* that is an example.

4.1.7 The answer does not contain solution snippets.

Even though a *how-to* question asks a specific programming question, expecting the other Stack Overflow users to post answers with *Solution Snippets*, some answers may not contain *Solution Snippets*. This category corresponds to such answers. There are two main reasons for an answer not to contain *Solution Snippets*. First, **the**

answer indicates there is no solution to the question. For example, in the question *"How do I disable right click on my web page?"*, the second-top-scored answer states that disabling right-clicks on a web page is not possible. Second, **the text in the answer describes how to solve the problem, but the answer does not provide a *Solution Snippet*** (neither within a code block nor inline with text). For example, the accepted answer to the question *"Best way to detect when a user leaves a web page?"* is shown in Figure 4.8. It explains which event handler in JavaScript can be used (`onbeforeunload`) to solve this question, but it does not contain a complete *Solution Snippet*.

> Try the `onbeforeunload` event: It is fired just before the page is unloaded. It also allows you to ask back if the user really wants to leave. See the demo *onbeforeunload Demo*.
>
> Alternatively, you can send out an Ajax request when he leaves.

Figure 4.8: An answer that explains how to solve the question without providing a *solution snippet*.

4.1.8 The answer contains solution snippets for related problems.

Sometimes answers contain at least one *Solution Snippet* for a programming question that is not explicitly asked in the original question. This category corresponds to such answers. Figure 4.9 shows the body of the question *"How can I get last characters of a string?"* The author of this question provides an example string "`ctl03_Tabs1`" in the question's body. The last code block in the accepted answer to this question is shown in Figure 4.10. The author of this answer suggests that the actual problem may be *retrieving the characters after the "_"*; they provide a *Solution Snippet* to solve it.

```
var id="ctl03_Tabs1";
```

Using JavaScript, how might I get the last five characters or last character?

Figure 4.9: The body of the question *"How can I get last characters of a string?"*

If you're simply looking to find the characters after the underscore, you could use this:

```
var tabId = id.split("_").pop(); // => "Tabs1"
```

Figure 4.10: The *solution snippet* that solves the problem *"getting the characters after the underscore in a string"* which was not asked in the original question (in Figure 4.9.

4.2 RQ2. How are solution snippets potentially adapted in order to be reused?

To answer RQ2, this study produced a categorization of the potential reusability aspects of *Solution Snippets*. There are five categories and are shown in Table 4.2. In this section, I describe each category in detail.

4.2.1 As-is

This category of reusability corresponds to the *Solution Snippets* that only need to be copied-and-pasted and do not need any customization. For example, the first *Solution Snippet* in the accepted answer to the question *"Is there a Newline constant defined in Java like Environment.Newline in C#?"* is shown in Figure 4.11. This *Solution Snippet* does not require any customization, such as additions and deletions of variables, data, or lines of source code when reusing the *Solution Snippet*.

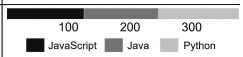
```
System.lineSeparator()
```

Figure 4.11: An *as-is solution snippet* which requires no customization in order to be reused.

4.2.2 User-defined functions (UDF)

This category of reusability corresponds to *Solution Snippets* that are presented as entire functions. In this thesis, such functions are called *User-defined functions* (UDF). The *Solution Snippet* is expected to be copied from Stack Overflow and pasted to the target source file. In order to adapt this *Solution Snippet*, the developer reusing it needs to write a function call to this UDF. For example, a code block in the accepted answer to the question *"How to list the properties of a JavaScript object?"* is shown in Figure 4.12. It contains a UDF named `getKeys`. In order to adapt this UDF to the target source code, the developer has to write a function call to `getKeys` with a suitable function argument for `obj`.

Table 4.2: The answer to RQ1: How *Solution Snippets* are potentially adapted in order to be reused

Categories	**Frequency of answers**	**Distribution of frequencies**
1. As-is: The *Solution Snippet* does not require any changes to be adapted to the target source code, and it can be reused as-is with a simple copy-and-paste.	77	
2. User-defined functions: The *Solution Snippet* is presented as an entire function (a user-defined function) — that is ready to be reused but requires a function call to be adapted to the target source code.	82	
3. Placeholders: The *Solution Snippet* contains sections that need to be completely removed or replaced in order to be adapted to the target source code.	95	
4. Customizable variables and/or data: The *Solution Snippet* contains variables and/or data that require customization to be adapted to the target source code.	375	
5. Conceptual: The *Solution Snippet* exemplifies a certain concept in the programming language. Such a *Solution Snippet* is not straightforward to be adapted to the target source code, as it must be understood and re-written from scratch.	69	

```
var getKeys = function(obj){
   var keys = [];
   for(var key in obj){
      keys.push(key);
   }
   return keys;
}
```

Figure 4.12: A *UDF*. The reuse of this snippet requires making a call to this UDF — `getKeys`.

4.2.3 Placeholders

This category of reusability corresponds to sections in *Solution Snippets* that must be replaced when they are adapted to target source files. In this thesis, such sections are called *placeholders*. Figure 4.13 shows a *Solution Snippet* in the accepted answer to the question *"How to include js file in another js file?"* This *Solution Snippet* contains the comment "`put your dependent JS here`." It needs to be replaced with the source code corresponding to the target source file. Two main types of placeholders are identified in this study:

Explicit placeholders. These placeholders convey messages that need to be interpreted and accordingly adapted by the developer. There are several types of such placeholders. First, some explicit placeholders are **comments that indicate they must be replaced when adapting *Solution Snippets***. For example, the comment "`put your dependent JS here`" in Figure 4.13 is an explicit placeholder which indicates that it must be replaced with the dependent javascript code snippet.

```
// jQuery
$.getScript('/path/to/imported/script.js', function()
{
    // script is now loaded and executed.
    // put your dependent JS here.
});
```

Figure 4.13: A *Solution Snippet* that contains a comment as a placeholder.

Second, some explicit placeholders are **print statements that indicate they need to be replaced** when adapting *Solution Snippets*. Figure 4.14 shows the *Solution Snippet* in the accepted answer to the question *"Correct way to pause a Python program."* This *Solution Snippet* has a print statement —

`print("something")` — that needs to be replaced with some code snippet when adapting the *Solution Snippet*.

```
import time
print("something")
time.sleep(5.5)    # Pause 5.5 seconds
print("something")
```

Figure 4.14: A *Solution Snippet* that contains print statements as placeholders.

Third, some explicit placeholders are **calls to the `alert` function (a javaScript-specific placeholder) that indicate they need to be replaced** when adapting *Solution Snippets*. The usage of the `alert` function call as an explicit placeholder is exemplified in Figure 4.15. This is the *Solution Snippet* in the accepted answer to the question *"How to check if a variable is an integer in JavaScript?"* The calls to `alert` function in this *Solution Snippet* need to be replaced with the appropriate code snippets.

```
if (data === parseInt(data, 10))
    alert("data is integer")
else
    alert("data is not an integer")
```

Figure 4.15: A *Solution Snippet* that contains calls to `alert` function as placeholders.

Lastly, other explicit placeholders are **text phrases within *Solution Snippets* that communicate those phrases must be replaced when adapting the *Solution Snippets***. For example, Figure 4.16 shows a *Solution Snippet* in the accepted answer to the question *"iterating over and removing from a map."* In this *Solution Snippet*, `<boolean expression>` is a placeholder in which the angle brackets (`<>`) encapsulate the text that communicates how it needs to be customized.

```
map.entrySet().removeIf(e -> <boolean expression>);
```

Figure 4.16: Using angle brackets (`<>`) to encapsulate the text that indicates that it needs to be customized.

Implicit placeholders. These placeholders suggest that they need to be replaced, but they have not explicitly communicated that. There are several types of such

placeholders. First, an **ellipsis** (...) is an implicit placeholder that does not explicitly communicate that it needs to be replaced. For example, Figure 4.17 shows a *Solution Snippet* in the second-top-scored answer to the question *"How to check if function exists in JavaScript?"* This *Solution Snippet* has an ellipsis between the curly braces which needs to be replaced with source code when adapting the *Solution Snippet*.

```
if (obj && typeof obj === 'function') { ... }
```

Figure 4.17: A *Solution Snippet* that contains an ellipsis (...) as a placeholder.

Second, Java's `Thread.sleep()` statement is an implicit placeholder. The usage of `Thread.sleep()` as a placeholder is exemplified in Figure 4.18, which is the top-scored answer to the question *"How to measure execution time for a Java method?"* In this *Solution Snippet*, `Thread.sleep(63553)` is an implicit placeholder because it does not explicitly indicate that it needs to be replaced. However, it has to be replaced with a code snippet in which the execution time needs to be measured.

```
Instant start = Instant.now();
Thread.sleep(63553);
Instant end = Instant.now();
System.out.println(Duration.between(start, end));
```

Figure 4.18: A *Solution Snippet* that contains a `Thread.sleep()` statement as a placeholder.

Lastly, a `pass` statement in Python is an implicit placeholder. For example, the *Solution Snippet* in the accepted answer to the question *"Most pythonic way to delete a file which may not exist"* is shown in Figure 4.19. The `pass` statement in this *Solution Snippet* needs to be replaced with appropriate source code by the developer in order to adapt this *Solution Snippet* to the target source file.

```
try:
    os.remove(filename)
except OSError:
    pass
```

Figure 4.19: A *Solution Snippet* that contains a `pass` statement as a placeholder.

4.2.4 Customizable variables and/or data

This category of reusability corresponds to *Solution Snippets* that contain *customizable variables and/or data*. When adapting such *Solution Snippets*, the customizable variables and/or data must be replaced with the corresponding variable names and/or data in the target source code. Figure 4.20 shows the code block with the *Solution Snippet* in the accepted answer to the question *"Is it possible to add dynamically named properties to JavaScript object?"* This code block contains an object named `data` that has the following properties: `PropertyA`, `PropertyB`, and `PropertyC`. The new property that needs to be added to this object is `PropertyD`. The *Solution Snippet* is `data["PropertyD"]=4`. Thus, when adapting this *Solution Snippet*, the object `data` must be replaced with the corresponding name of the object that needs to be extended; the string "`PropertyD`" must be replaced with the name of the new property to be added to the object; and the number `4` must be replaced with the value of the new property.

```
var data = {
    'PropertyA': 1,
    'PropertyB': 2,
    'PropertyC': 3
};

data["PropertyD"] = 4;

// dialog box with 4 in it
alert(data.PropertyD);
alert(data["PropertyD"]);
```

Figure 4.20: A *solution snippet* with customizable variables and/or data.

4.2.5 Conceptual

This category of reusability corresponds to *Solution Snippets* that are not intended to be copied-and-pasted and adapted for reuse: they are examples of concepts rather than reusable code. Furthermore, the sixth category described in the previous section (Section 4.1.6) corresponds to this category of reusability. For example, the answers to the question *"How to pass a function as a parameter in Java?"* contain *Solution Snippets* that exemplify the programming concept of passing a function to another function as a parameter. The *Solution Snippet* in the second top-scored answer (shown in Figure 4.21) to this question is not directly reusable. This *Solution*

Snippet spans the entire code block. Its `method1` and `method2` are arbitrary methods used to illustrate the concept. The developer has to read this *Solution Snippet*, understand the underlying programming concept, and apply it to the methods in the target source code.

```
import java.lang.reflect.Method;

public class Demo {

    public static void main(String[] args) throws Exception{
        Class[] parameterTypes = new Class[1];
        parameterTypes[0] = String.class;
        Method method1 = Demo.class.getMethod("method1", paramet

        Demo demo = new Demo();
        demo.method2(demo, method1, "Hello World");
    }

    public void method1(String message) {
        System.out.println(message);
    }

    public void method2(Object object, Method method, String mes
        Object[] parameters = new Object[1];
        parameters[0] = message;
        method.invoke(object, parameters);
    }

}
```

Figure 4.21: A Conceptual *solution snippet*.

4.3 Summary

This study produced two categorizations. The first categorization is how *Solution Snippets* are presented in answers posted on Stack Overflow, and there are eight categories. These categories describe where *Solution Snippets* are found in Stack Overflow (either inline or within code blocks), the number of *Solution Snippets* in the answer, what type of ancillary code is found within code blocks in addition to *Solution Snippets*, etc. The second categorization is how *Solution Snippets* need to be adapted for reuse, and there are five categories. These categories describe what parts of a *Solution Snippet* need to be adapted by a developer who reuses it: for example, whether the *Solution Snippet* could be reused as-is or whether the *Solution Snippet* could be reused after customizing the variables and data, or whether

the *Solution Snippet* contains placeholders that need to be replaced with any code snippets according to the target source code, etc.

5

Comparison to Other Research

The following three studies are closely related to this study. First, Nasehi et al. [34] identified the categories of recognized answers[1] in Stack Overflow. Second, Zagalsky et al. [55] identified how *Solution Snippets* are described in answers in Stack Overflow. Lastly, Wu et al.'s [49] studied how Stack Overflow's code blocks are reused in GitHub. In this chapter, I discuss how the results of my study are related to the studies mentioned above.

Nasehi et al. [34] identified nine categories of recognized answers in Stack Overflow; their categories are: *Concise Code*, *Using Question Context*, *Step-by-Step solution*, *Multiple solutions*, *Highlighting Important Elements*, *Providing Links to Extra Resources*, *Inline Documentation*, *Solution Limitations*, and *API Limitations*. The first four categories apply to code blocks in answers, while the remaining categories apply the text in answers. Nasehi et al. considered that a *Solution Snippet* spans an entire code block. To describe features of code blocks, they used the number of code blocks in an answer and the availability of source code comments. In contrast, the study described in my thesis is more fine-grained, as it identifies *Solution Snippets* within code blocks. Nasehi et al.'s first category (*Concise Code*) also identified an aspect of reusability: the *placeholders*. In contrast, the categorization of reusability aspects of *Solution Snippets* described in my thesis (Table 4.2) is more comprehensive than Nasehi et al.'s.

Zagalsky et al. [55] identified nine categories of Stack Overflow's answers: *Explanation*, *Redirecting*, *Clue/Hint/Suggestion*, *Alternative*, *Tutorial*, *Announcement*, *Opinion*, *Benchmark*, and *Source code*. Except for the last category (*Source code*),

[1] In [34], a recognized answer is an answer that is either an accepted answer or an unaccepted answer with a normalized score of 0.4 or more.

Zagalsky et al.'s categories described how the text in answers explains the code blocks. As opposed to Zagalsky et al.'s categories, the ones described in my thesis are entirely focused on *Solution Snippets*: (1) how they are embedded within code blocks and text in Stack Overflow's answers (Table 4.1) and (2) how they need to be adapted for reuse (Table 4.2).

Wu et al. [49] identified five categories of how Stack Overflow's code blocks are reused in projects found in GitHub. These categories are: *Exact Copy*, *Converting Ideas*, *Cosmetic Modification* (the modifications that do not change the functionality of a code snippet), *Non-Cosmetic Modification* (the modifications that change the functionality of a code snippet), and *Providing Information*[2]. The study described in my thesis adds a different perspective to Wu et al.'s study, and the two studies are complementary to each other: Wu et al. described how developers had adapted Stack Overflow's code blocks, while my thesis describes how *Solution Snippets* within Stack Overflow's code blocks could be potentially adapted for reuse. In my thesis, the syntax and semantics of the *Solution Snippets* are used to describe the categories, and thus, they are more fine-grained than Wu et al.'s categories. Wu et al.'s first two categories — *Exact Copy* and *Converting Ideas* — are similar to two categories described in my thesis: *As-is* and *Conceptual*, respectively. Wu et al.'s category — *Non-cosmetic Modification* — is similar to the combination of the following three categories described in my thesis: *User-defined functions*, *Placeholders* and *Customizable variables and/or data*. As informed by the names of the categories, Wu et al.'s categories are more specific about how similar the source code in GitHub is to the code blocks in a Stack Overflow's answer, while the categories described in my thesis are more specific about which programming constructs in the *Solution Snippet* are required to be adapted for reuse. Lastly, according to my thesis, a *Solution Snippet* could have multiple aspects of reusability, while according to Wu et al.'s study, only one of their categories is applicable to any source code found on GitHub that reused code blocks in Stack Overflow's answers.

In summary, the two categorizations presented in this study further expand some of the prior studies ([34, 55]) and complement one other ([49]).

[2]Stack Overflow's answers that are used as references.

6

Discussion

During the first part of my study, I made several observations about Stack Overflow's questions, and I propose that these observations might be useful to improve the methods to create datasets of Question-Solution pairs. I begin this chapter by discussing these observations (Section 6.1). During the second part of my study, I identified answers to the two research questions. By analyzing them, I next discuss what researchers need to be aware of in order to improve the methods to create datasets of Question-Solution pairs: the potential reasons why it is challenging to find and extract *Solution Snippets* (Section 6.2), and the challenges of adapting them for reusing (Section 6.3). Then, I discuss what Stack Overflow could do to help researchers avoid some of these challenges (Section 6.4). I end this chapter by describing the threats to the validity of my study (Section 6.5).

6.1 Observations made about Stack Overflow's questions

I made several observations about *how-to* questions in Stack Overflow. First, the title of a *how-to* question often begins with a phrase, such as *"how to," "how can," "can I,"* and *"is it possible to,"* follows by a specific programming problem. For example, the question *"How to convert Set<String> to String[]?"* begins with the phrase *"how to,"* and it is followed by the specific programming problem *"convert Set<String> to String[]."* Second, in order to create concise titles, authors of some *how-to* questions did not phrase titles as questions. For example, the question *"Find the min/max element of an Array in JavaScript"* does not begin with a question phrase, but it

contains a specific programming problem. Such questions in Stack Overflow are also classified as *how-to* questions in my study.

Among the questions that were not classified as *how-to* questions in my study, the following are the most frequent types of questions I observed. Such questions are not useful to create Question-Solution pairs [28, 53].

- A question may contain a phrase like in a *how-to* question (*"how to," "can I,"* etc.), but it may not contain a specific programming problem. For example, the question *"How to execute a java .class from the command line?"* contains the phrase *"how to"* in its title. This question does not ask for a *Solution Snippet* but a shell command, and it is not considered as a *how-to* question that is useful to create a Question-Solution pair.

- A question may ask for an explanation. For example, the python question *"What is the source code of the 'this' module doing?"* asks for an explanation, and it is not a *how-to* question.

- A question may ask for listing the differences between two or more programming constructs. For example, the question *"java: Class.isInstance vs Class.isAssignableFrom"* asks for differences, and it is not a *how-to* question.

6.2 Why is it challenging for automatic extractors to find and extract *Solution Snippets*?

Stack Overflow continues to grow without any specific constraints on how questions should be asked or answered, and its only restriction is that a question should be a programming question. While Stack Overflow provides recommendations[1] on how to answer questions (for example, providing context for links and being polite), it is mainly up to the author of an answer how a *Solution Snippet* is presented. The results of RQ1 (the categories of how *Solution Snippets* are presented in answers in Section 4.1) imply two potential reasons why a *Solution Snippet* is not easy to be found and extracted:

- It is not easy to find a *Solution Snippet*: frequently, answers contain multiple code blocks, and not all of them have *Solution Snippets*; in some cases, the *Solution Snippet* is inlined with text.

[1] https://stackoverflow.com/help/

- It is challenging to separate a *Solution Snippet* from the non-relevant source code within a code block: *Solution Snippets* are often surrounded by ancillary code, and in some cases, the *Solution Snippet* is split by ancillary code.

The following two subsections describe the above in detail.

6.2.1 It is not easy to find *Solution Snippets*.

Stack Overflow's Markdown dialect supports tagging source code as separate code blocks and inline code with text. Thus, there are two potential locations where *Solution Snippets* may be found in an answer: **within code blocks** and **inline with text**.

When an answer **contains only one code block, this code block usually contains the *Solution Snippet*** (46.3% of the answers — 277 out of 598 — contained only one code block, and of them, 93.1% contained only one *Solution Snippet* within the code block). If automatic extractors only use such answers to create datasets of Question-Solution pairs, it is likely to reduce the number of pairs extracted from Stack Overflow to nearly 50% of the potential pairs.

When an answer **contains multiple code blocks as well as inline code snippets, it is not easy to find which code blocks and/or inline code contain *Solution Snippets* and discard the rest** (of the answers studied in this research, 53.7% — 321 out of 598 — contained multiple code blocks and 48.0% — 287 out of 598 — contained both code blocks and inline code snippets). For example, Figure 6.1 shows the first three code blocks of the accepted answer to the question *"How to return the product of a list?"*. This answer contains five code blocks and two inline code snippets. Of them, only the first two code blocks (as shown in the figure) contain *Solution Snippets*. The first code block contains a *Solution Snippet* that spans the entire code block, while the second code block contains four *Solution Snippets* in which the lines of source code are interleaved with one another (1. lines a & e, 2. line d, 3. lines b & f, and 4. lines c & g). The remaining three code blocks and the two inline code snippets do not contain any *Solution Snippets*. Furthermore, the third code block contains sample inputs; the last two code blocks contain execution time-related information; the two inline code snippets contain identifier names. For this answer, the assumption that every code block contains a *Solution Snippet* is incorrect.

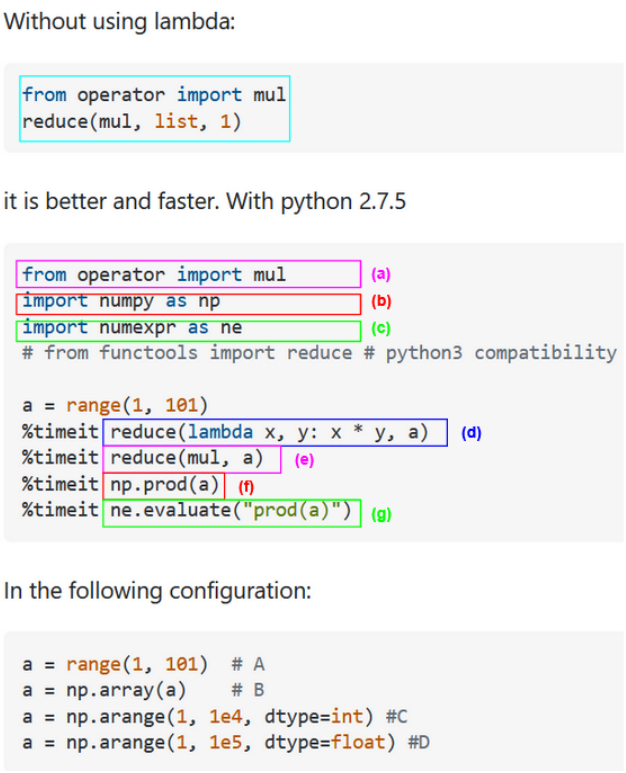
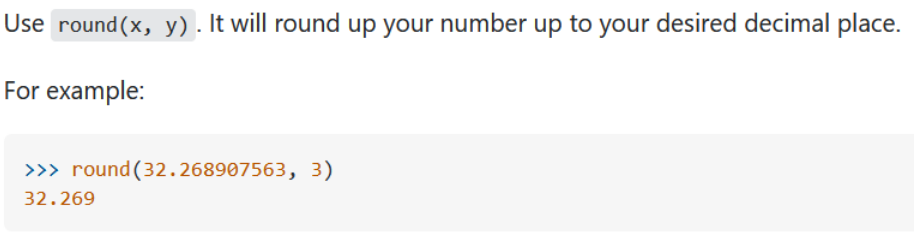


Figure 6.1: The first three code blocks of an answer to a Python question. The *Solution Snippets* are found in the first two code blocks.

When a *Solution Snippet* is a **library function call, it is frequently presented as an inline code snippet**. Of the answers studied in this research, 9.5% (57 out of 598) contained inline *Solution Snippets*. Of them, 71.9% were simply library function calls. For example, Figure 6.2 shows the second top-scored answer to the question *"How to round a number to the nearest integer?"*. This answer contains an inline code snippet and a code block. The inline code snippet contains the *Solution Snippet* `round(x,y)`, and the code block exemplifies the usage of the *Solution Snippet*.

A code block may **contain multiple *Solution Snippets***. For extracting such *Solution Snippets*, the automatic extractors have to identify the number of *Solution Snippets* within the code block and the boundaries of each *Solution Snippet* (5.0% of the answers — 30 out of 598 – contained at least one code block with more than one *Solution Snippet*). For example, the second code block in Figure 6.1 contains four *Solution Snippets* to the question *"How to return the product of a list?"*. Considering the entire code block as a single *Solution Snippet*, in this case, is not accurate.

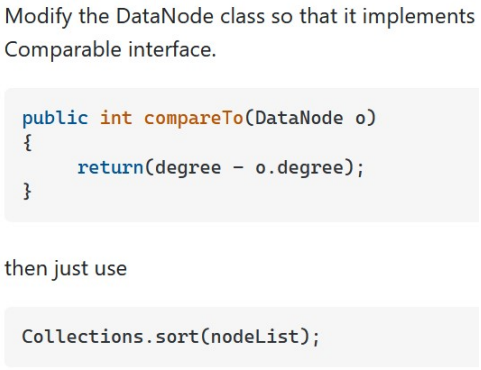

Use `round(x, y)`. It will round up your number up to your desired decimal place.

For example:

```
>>> round(32.268907563, 3)
32.269
```

Figure 6.2: An answer to a Python question. This answer contains an inline code snippet as well as a code block. The inline code snippet is a *Solution Snippet*, and the code block contains an example and its output.

In addition, **one *Solution Snippet* may span over multiple code blocks**. There were 18 such *Solution Snippets* across 14 answers among the ones I studied. Even though such *Solution Snippets* are infrequent, they highlight the challenges of extracting *Solution Snippets* from Stack Overflow. In such cases, the text in the answer may contain contextual information that helps understand how a *Solution Snippet* should be reused. For example, the answer to the question *"Sort an ArrayList based on an object field"* is shown in Figure 6.3. It contains two code blocks: (1) the first code block contains a user-defined function (`compareTo`) that compares two objects based on a specific field (`degree`), and (2) the second code block contains the source code that sort an `ArrayList`. Moreover, the text in the answer indicates that the class of the objects in the `ArrayLst` should be modified to reuse the *Comparable* Java interface, and the user-defined function in the first code block should be written within that class. The source code in the second code block should be reused where the sorted `ArrayLst` is required. In this case, considering each code block as a *Solution Snippet* is inaccurate.


Modify the DataNode class so that it implements Comparable interface.

```
public int compareTo(DataNode o)
{
     return(degree - o.degree);
}
```

then just use

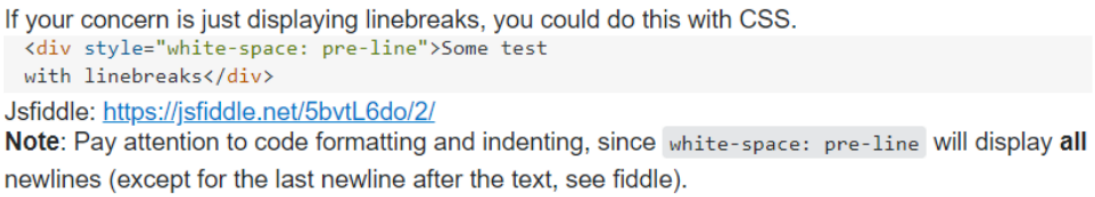
```
Collections.sort(nodeList);
```

Figure 6.3: An answer to a Java question. It contains two code blocks, and one *Solution Snippet* spans over these two code blocks.

In some answers, *Solution Snippets* may be labelled with programming language versions and library versions. Of the answers, 13.9% (83 out of 598) contained *Solution Snippets* labelled with a specific programming language version, while 3.8% (23 out of 598) contained *Solution Snippets* labelled with a specific library version. Because the *Solution Snippets* labelled with one version may exhibit compatibility issues with another version, the automatic extractors may have to identify such *Solution Snippets* and incorporate the information on versions also into the datasets of Question-Solution pairs. Irrespective of being labelled with a version or not, some *Solution Snippets* may be outdated. Such outdated *Solution Snippets* may exhibit security vulnerabilities or might have been superseded by better *Solution Snippets* [41]. Having such *Solution Snippets* may degrade the quality of datasets of Question-Solution pairs.

An answer **may contain *Solution Snippets* that solve a different question** (instead of the original question indicated in the title). For example, Figure 4.10 (in Section 4.1) shows the last code block in the accepted answer to the question *"How can I get last characters of a string?"*. It contains a *Solution Snippet* to another question, *"How to find the characters after the underscore?"* (one that the author does not explicitly ask in the original question). In such cases, *Solution Snippets* that answer other questions must be discarded when extracting a *Solution Snippet* to the original question.

Some answers that have high scores **do not contain any *Solution Snippets***. There were 4.3% (26 out of 598) of such answers. Figure 6.4 shows the second top-scored answer to the question *"How do I replace all [the] line breaks in a string with
 elements?"*. It contains a code block and one inline code snippet, but both are in HTML/CSS instead of JavaScript, as the question requested. It is not possible to create a Question-Solution pair using this answer at all, and labelling either the code block or inline code snippet as a *Solution Snippet* in this answer is incorrect.


If your concern is just displaying linebreaks, you could do this with CSS.

```
<div style="white-space: pre-line">Some test
with linebreaks</div>
```

Jsfiddle: https://jsfiddle.net/5bvtL6do/2/

Note: Pay attention to code formatting and indenting, since `white-space: pre-line` will display **all** newlines (except for the last newline after the text, see fiddle).

Figure 6.4: An answer to a Javascript question without Javascript source code in its code block. The answer contains a code block and an inline code snippet, but both are in CSS.

Lastly, to create datasets of Question-Solution pairs, some automatic extractors (e.g. [28, 53, 54]) paired the *Solution Snippets* in the accepted answer to questions' titles, but **the score of an answer appears to be a better indicator when selecting answers**. In fact, the accepted answer for a question may not be the top-scored answer to the question, and this implies that the answer selected by the author of the question to be the "accepted answer" might not be the best. The accepted answer was not the top-scored answer in 26% (78 out of 300) of the questions studied in this research.

6.2.2 It is challenging to separate a *Solution Snippet* from the ancillary code within a code block.

Once a code block is identified as containing a *Solution Snippet*, the next problem is to separate the *Solution Snippet* from the ancillary code within the code block (when a *Solution Snippet* is found in an inline code snippet, it is usually the *Solution Snippet*, and no ancillary code is found). There were 29.5% of answers (177 out of 598) containing at least one code block that has a *Solution Snippet* with the following two cases of ancillary code that need to be extracted separately:

Case 1: Prologue and epilogue of a *Solution Snippet*. *The prologue of a Solution Snippet* is the source code within a code block that appears before the *Solution Snippet*. The two main uses of the prologue is for setting up the variables and for establishing the context for the *Solution Snippet* in use. *The epilogue of a Solution Snippet* is the source code that appears after the *Solution Snippet*, which exemplifies the *Solution Snippet*'s usage, often showing its output. Prologues and epilogues are not part of the *Solution Snippet*, but help understand how a *Solution Snippet* works. For example, Figure 6.5 shows a code block in the second top-scored answer to the question *"How to [use] for each [with] the hashmap?"*. Its prologue contains the source code to set up (instantiate and populate) the hashmap, while its epilogue contains the output generated by the *Solution Snippet*.

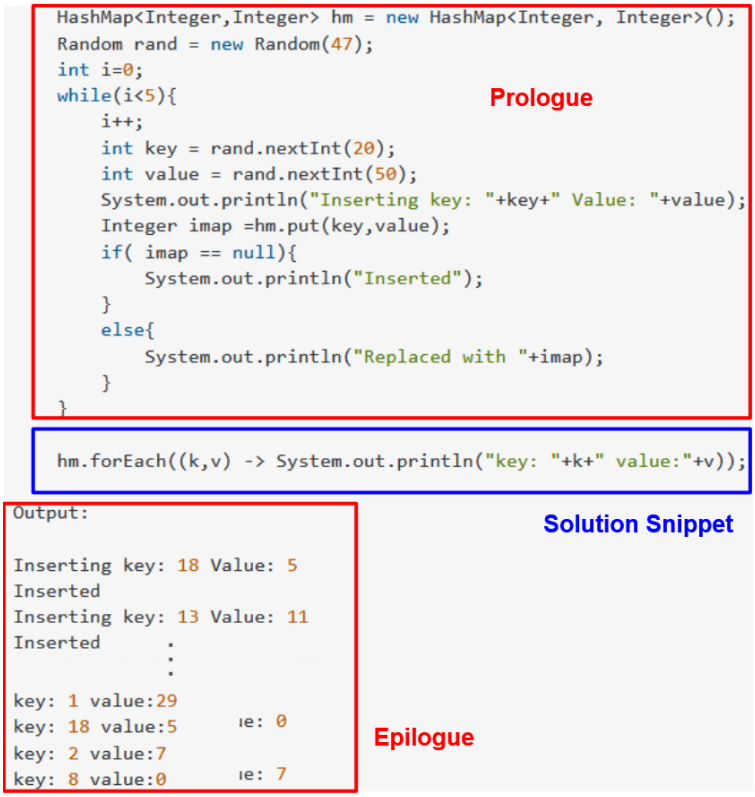


Figure 6.5: A code block that contains a prologue, a *solution snippet*, and an epilogue.

Some code blocks contain much simpler prologues and epilogues. For example, Fig. 6.6 shows the code block in the second top-scored answer to the question *"How can I convert a long to [an] int in Java?"*. The *Solution Snippet* in this code block is annotated in blue. The rest of the source code just before the *Solution Snippet* is the prologue. It contains a sample `Long` input `x` and variable `y` that holds the `int` output of the *Solution Snippet*. Even though the prologue is much simpler, it establishes the context for the *Solution Snippet*, providing some information about the input and output variable types of the *Solution Snippet*. However, the input data (`100L`) is not specific to the question asked in the title, and the prologue is not part of the *Solution Snippet*.

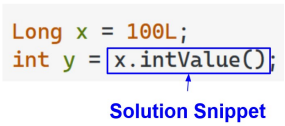


Figure 6.6: A code block that contains a much simpler prologue that establishes the context for the *Solution Snippet* in use.

Case 2: Interleaved ancillary code. A *Solution Snippet* could be composed of two or more non-contiguous code snippets interleaved with ancillary code. When extracting such a *Solution Snippet*, the ancillary code interleaved the code snippets in the *Solution Snippet* must be identified and extracted separately. For example, Figure 6.7 shows the code block in the accepted answer to the question *"How do I get a list of methods in a Python class?"*. This code block contains two *Solution Snippets*: the first one is composed of contiguous snippets (a) and (b) (excluding the comment "`#python2`"), and the second one is composed of non-contiguous snippets (a) and (c). This code block contains two import statements. One import statement (for the package `inspect`) is part of both the *Solution Snippets*, while the other import statement (for the package `optparse`) imports a sample package to get a class (`OptionParser`) to demonstrate the use of these *Solution Snippets*.

```
>>> from optparse import OptionParser
>>> import inspect   Snippet (a)
#python2                                                    Snippet (b)
>>> inspect.getmembers(OptionParser, predicate=inspect.ismethod)
[([('__init__', <unbound method OptionParser.__init__>),
...
 ('add_option', <unbound method OptionParser.add_option>),
 ('add_option_group', <unbound method OptionParser.add_option_group>),
 ('add_options', <unbound method OptionParser.add_options>),
 ('check_values', <unbound method OptionParser.check_values>),
 ('destroy', <unbound method OptionParser.destroy>),
 ...
 ]
# python3                                                   Snippet (c)
>>> inspect.getmembers(OptionParser, predicate=inspect.isfunction)
...
```

Figure 6.7: A code block that contains two *solution snippets*. The first *solution snippet* is formed by snippets (a) and (b), and the second *solution snippet* is formed by non-contiguous snippets (a) and (c).

The case of the first *Solution Snippet* is similar to Case 1 I discussed before. In particular, this *Solution Snippet*'s prologue is the import statement for the `optparse`, and its epilogue is all the code snippets below the snippet (b), including REPL outputs and the snippet (c) of the second *Solution Snippet*.

The case of the second *Solution Snippet* is Case 2. Its prologue contains the import statement for the `optparse`, REPL outputs, and the snippet (b). The code snippets in this prologue interleave the lines of source code in the *Solution*

Snippet, dividing it into two sections. An automatic extractor first needs to remove outputs from the REPL and then separately extract the *Solution Snippet* and the code snippets of the prologue that interleaved this *Solution Snippet*.

Some answers may contain prologues and epilogues in separate code blocks other than the ones containing *Solution Snippets*. Nevertheless, if a *Solution Snippet* is surrounded by ancillary code within a code block, the *Solution Snippet* and the ancillary code must be tagged correctly and/or extracted separately when creating datasets.

6.3 What are the potential difficulties of adapting Stack Overflow's *Solution Snippets*?

To generate a *Solution Snippet* adapted to a target source code, machine learning models may require the understanding of how *Solution Snippets* need to be adapted for reuse. According to the results of RQ2 in Section 4.2, there are five categories of how *Solution Snippets* need to be adapted for reuse (the potential reusability aspects of *Solution Snippets*). Of them, ***User-defined function* and *As-is Solution Snippets* are easy to reuse, while the remaining three categories are not**: *Customizable-variables and/or data*, *Placeholders*, and *Conceptual*.

As-is Solution Snippets could be reused by copying and pasting without modifying the source code. Perhaps the only customization these *Solution Snippets* require is to adhere to the project's coding standards, such as renaming identifiers and editing white spaces. A *User-defined function* could be reused by copying and pasting it in the target source code and calling it with function arguments. In some answers, how to do this is illustrated in the epilogue of the *Solution Snippet*.

Customizable-variables and/or data require replacing sections of a *Solution Snippet* with the source code that adapts the *Solution Snippet* to a particular use. In order to understand how the *Solution Snippet* is potentially reused, it may be required to understand the prologue and/or the epilogue of the *Solution Snippet*. For example, as shown in Figure 6.5, the contextual information that is needed to reuse the *Solution Snippet* is provided in the prologue: what `hm` is and how it needs to be instantiated.

The code blocks presented within the question's body may provide context to *Solution Snippets* in its answers, and it is challenging to understand how such *Solution Snippets* need to be adapted for reuse. There were 31.6% of the answers (189 out of 598) containing *Solution Snippets* that reuse identifiers, source code or data presented

in the question's code blocks. Because it may require analyzing questions' code blocks in order to reuse such *Solution Snippets*, simply storing the question's title in a Question-Solution pair— as done in existing methods to create datasets of Question-Solution pairs ([4, 28, 53, 54]) — may not be sufficient; it may also require storing the question's code blocks. For example, the body of the question *"How to get [the cuurent] Month [using] JavaScript in 2 digit format?"* is shown in Figure 6.8, and the second *Solution Snippet* in the accepted answer to this question is shown in Figure 6.9. In this answer, it is not explicitly stated what the keyword `this`[2] in the *Solution Snippet* is referred to. The keyword `this` in JavaScript refers to the current object. According to the question's body, the keyword `this` in this particular *Solution Snippet* has to be referring to a `Date` object.

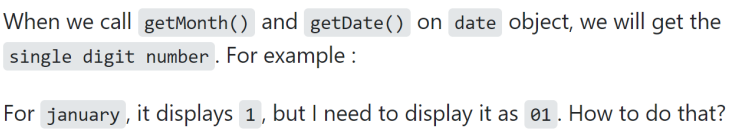


Figure 6.8: The body of the question *"How to get [the cuurent] Month [using] JavaScript in 2 digit format?"*

```
("0" + (this.getMonth() + 1)).slice(-2)
```

Figure 6.9: A *Solution Snippet* containing a reference (`this`) to an object that is not defined within any code block in the answer.

For another example, Figure 6.10 shows a code block found within the question *"How to sort [an] array by [the] firstname (alphabetically) in Javascript?"*. This code block contains the implicit information that the array should comprise objects with a field `firstname`. Such information is useful for adapting the *Solution Snippets*. Figure 6.11 shows the *Solution Snippet* in the accepted answer to this question. This *Solution Snippet* spans the entire code block, and it uses an array (`users`) that is not defined within its code block. The structure of an object (`user`) used in this array is found in the question's code block shown in Figure 6.10.

[2] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this`

```
var user = {
   bio: null,
   email:  "user@domain.com",
   firstname: "Anna",
   id: 318,
   lastAvatar: null,
   lastMessage: null,
   lastname: "Nickson",
   nickname: "anny"
};
```

Figure 6.10: A code block in a question

```
users.sort(function(a, b){
    if(a.firstname < b.firstname) { return -1; }
    if(a.firstname > b.firstname) { return 1; }
    return 0;
})
```

Figure 6.11: A *Solution Snippet* having an array that is not defined within any code block in the answer.

Some *Solution Snippets* contain *placeholders* that need to be replaced to adapt the *Solution Snippet* to the target source code. Identifying placeholders is not trivial because several types of placeholders are used, and there is no standard convention of what a placeholder should look like. As mentioned in Section 4.2, there are two main types of placeholders. The first type is *explicit placeholders* which convey messages to the developer. Such messages need to be interpreted and adapted accordingly, such as print statements and comments (e.g. "`Your code goes here`"). The second type is *implicit placeholders* which need to be replaced but have not explicitly communicated that, such as ellipsis (`...`) and `pass` statements in Python. Some of these placeholders are language-specific (e.g. `pass` statements in Python), while others are language-agnostic (e.g. ellipsis). For example, the *Solution Snippet* in the top-scored answer to the question *"How can I check whether a radio button is selected with JavaScript?"* is shown in Figure 6.12. It contains two explicit placeholders (comments) that need to be replaced when adapting this *Solution Snippet*.

```
if(document.getElementById('gender_Male').checked) {
  //Male radio button is checked
}else if(document.getElementById('gender_Female').checked) {
  //Female radio button is checked
}
```

Figure 6.12: A *Solution Snippet* that has two placeholders.

For another example, Figure 6.13 shows the *Solution Snippet* in the accepted answer to *"How do I get [the] current index/key in [a] 'for each' loop?"* It contains a print statement, an explicit placeholder that needs to be replaced when adapting this *Solution Snippet*. This placeholder encloses a part of the *Solution Snippet* (`index++`), and thus, the placeholder is tightly integrated with the source code that must be kept when adapting this *Solution Snippet*.

```
int index = 0;
for(Element song : question) {
    System.out.println("Current index is: " + (index++));
}
```

Figure 6.13: A *Solution Snippet* having a print statement that is a placeholder. This particular placeholder encloses a part of the *Solution Snippet* (`index++`).

However, not all print statements are placeholders; if the question is about logging, a print statement in a *Solution Snippet* could be part of it; in such cases, a print statement is no longer a placeholder, and it might not need to be replaced when adapting the *Solution Snippet*. For example, a *Solution Snippet* in the top-scored answer to the question *"Print an integer in binary format in Java"* is shown in Fig. 6.14. This question is about printing; thus, the print statement in this *Solution Snippet* is not a placeholder.

```
System.out.println(Integer.toBinaryString(x));
```

Figure 6.14: A print statement that is part of the solution to the question

In some cases, the *Solution Snippet* contains code snippets that are supposed to be pasted in different places in the target source file. For example, a *Solution Snippet* in the second top-scored answer to the question *"How to calculate a logistic sigmoid function in Python?"* is shown in Figure 6.15. This *Solution Snippet* has an import statement (annotated in red) that usually needs to be included within the first few lines in a source file, while the remaining line of source code in the *Solution Snippet*— `logistic.cdf(0.458)` — needs to be pasted somewhere else in the source file and needs to be customized (by changing the input `0.458`).

The most challenging *Solution Snippets* to be reused are the *Conceptual Solution Snippets*. These *Solution Snippets* are examples illustrating programming concepts. Such *Solution Snippets* are not easy to simply copy-and-paste and customize. Almost all the source code in such a *Solution Snippet* needs to be replaced. For example, a

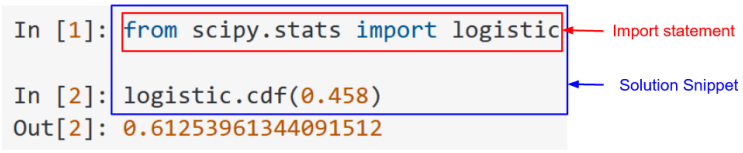


Figure 6.15: A *Solution Snippet* that has two sections: an import statement and the remaining source code in the *Solution Snippet*.

Solution Snippet in the second top-scored answer to the question *"How can I access 'static' class variables within methods in Python?"* is shown in Figure 6.16. This *Solution Snippet* spans over the entire code block, and it contains a class (`Foo`), a class variable (`bar`), and a *User-defined function* (`bah`) that are not ready to be reused. The class name, the class variable, the function, and the argument passed to the function are arbitrary. In addition, this *Solution Snippet* contains a *placeholder* (a print statement) illustrating how the class variable is accessed within the function body. Thus, the *Solution Snippet* itself cannot be reused without heavy customization, requiring almost all the source code to be replaced when adapting it.

```
class Foo(object):
    bar = 1
    @classmethod
    def bah(cls):
        print cls.bar
```

Figure 6.16: A conceptual *Solution Snippet*.

6.4 What Stack Overflow could do to help improve the automatic extraction of *Solution Snippets*.

Stack Overflow users utilize its Markdown dialect (1) to specify how the text is displayed and (2) to delimit code blocks and inline code snippets. It lets **users specify how the content looks but not the semantics of the code snippets within code blocks**, such as *Solution Snippets*, placeholders, examples, inputs, and REPL outputs. To encourage users to specify the semantics of code snippets, Stack Overflow could document some widely used conventions. Some of these conventions are language-agnostic, such as using ellipsis as placeholders, while others are language-specific, such as keeping the REPL prompt to separate Python source code from its output and specifying the language versions for which the *Solution Snippet* applies.

In addition, Stack Overflow could enhance its Markdown dialect for semantic tagging of code snippets by introducing new features, such as the following:

- A feature to indicate whether a code block contains a *Solution Snippet*.
- A feature to tag, within a code block, the *Solution Snippet*, its prologue, and its epilogue.
- A more structured way to add metadata to the *Solution Snippet*, such as a brief description of the problem it solves, the version of the programming language it uses, any libraries the *Solution Snippet* requires, and the information about how the *Solution Snippet* is expected to be reused (e.g., *as-is*, *placeholders*, etc.).

Stack Overflow partially moved in this direction when it added the support for *runnable*[3] JavaScript, HTML, and CSS code snippets. Its Markdown dialect lets the user specify a code block as *runnable snippets*, and the output of a code block is separated in such snippets. While this feature has been available since 2014, it is not widely used (of the JavaScript answers studied in this research, only 16.5% — 33 out of 200 — of the JavaScript answers contained *runnable* code snippets).

On the one hand, the suggested enhancements to Stack Overflow's Markdown dialect would help researchers extract *Solution Snippets* and the related information to create datasets of high-quality Question-Solution pairs. On the other hand, the suggested enhancements would benefit the developers who use Stack Overflow for day-to-day programming.

6.5 Threats to Validity

There are several threats to the validity of the results of my study. In this section, I summarize those threats and the precautions taken to mitigate them. For reproducibility and credibility purposes, I made the coded data available at [6].

Internal validity: Internal validity is the extent to which a researcher draws correct inferences from the data collected [17, 45]. In this study, there are three main threats to internal validity. First, the selection of Stack Overflow's questions and their answers could have incurred a *selection bias*. The questions were

[3]https://stackoverflow.blog/2014/09/16/introducing-runnable-javascript-css-and-html-code-snippets/

not randomly selected; instead, the top-scored questions in Stack Overflow were used. Because the most useful *Solution Snippets* will likely come from the answers to Stack Overflow's top-scored questions, I believe using the top-scored questions for this study is not a significant threat to the validity. Second, this study relies on human judgement when classifying Stack Overflow's questions and coding their answers, and thus, a certain amount of *subjectiveness* is expected [17]. To minimize the possible subjectiveness when classifying questions, three coders iteratively classified questions until they reached an agreement, as suggested in [21]. To minimize the possible subjectiveness when coding answers, the three coders iteratively coded the answers until the saturation of categories, as suggested in [17, 26]. Third, the coders studied answers by navigating to their URLs. During an iteration, Stack Overflow users may have edited some of the answers used in an iteration. Thus, different coders may have seen different versions of the same answer when coding. This risk was likely alleviated when the coders discussed their discrepancies at the end of an iteration.

External validity: External validity is the validity of applying conclusions derived from sample data to new subjects, new settings, and new situations [17]. The subjects of this study were the top-scored questions in Stack Overflow and their top-scored answers. Accordingly, there are several threats to the external validity of the results. First, only the questions and answers in Stack Overflow were studied. Because the structure of the questions and answers could be different in other Q&A platforms (Quora, Reddit, CodeProject, etc.), this study's results may not be applicable for those platforms. Second, only the questions and answers tagged with JavaScript, Java, and Python were studied. It produced language-agnostic results as well as language-specific results (e.g. showing the *Solution Snippet* executed in REPL along with outputs is specific for Python). Therefore, I believe that the results of this study may be extended to Stack Overflow's questions tagged with other programming languages by coding their answers while using the categories identified in this study as the initial codes. Third, the results are grounded in the sample of top-scored questions and their top-scored answers. The answers to questions with a low score may have different categories than those identified in this study. Overall, I do not claim the results of this study are generalized over the entire population of questions and answers in Stack Overflow and other Q&A platforms. Because a qualitative

study intends to provide results with respect to a specific context [17, 26], I believe the lack of generalizability is not a critical threat to the validity of this study's results.

Construct validity: Construct validity is the extent to which the variables of interest are adequately measured [17]. There are two threats to the construct validity of this study. First, at the beginning of this study, the coders had little agreement on how to identify *how-to* questions in Stack Overflow, and it was required to measure the consistency across the classification done by different coders [22]. In order to achieve a good agreement, the coders classified batches of questions, discussing the discrepancies until the *Fleiss Kappa coefficient* exceeded a threshold of 0.9, which is higher than the substantial strength of agreement suggested in [30]. Second, the coders stopped coding once they noticed that the categories had remained unchanged since the previous iteration. In order to ensure that the categories were saturated and premature closure of coding was avoided, I continued to code new answers (as suggested in [2, 5]). I stopped coding after I coded a satisfactory amount of new answers and became convinced that the categories were saturated.

6.6 Summary

After reflecting on the results of RQ1, I identified several potential reasons why it is challenging to find and extract *Solution Snippets* automatically from Stack Overflow. A *Solution Snippet* may be found in either code blocks or inline with the text, and it is required to scan through all the code blocks and inline code snippets to identify which of them contain *Solution Snippets*. If the *Solution Snippet* is inside a code block, separating it from the ancillary code within the code block is challenging.

After reflecting on the results of RQ2, the *Solution Snippets* showing the first two reusability aspects (*User-defined function* and *As-is Solution Snippets*) appear to be easy to reuse while the *Solution Snippets* showing the rest of the reusability aspects do not (*customizable-variables and/or data*, *Placeholders*, and *Conceptual*). There are several types of placeholders, and there is no consistency between them; they need to be identified and replaced with specific source code when adapting *Solution Snippets*. Customizable variables and/or data also need to be identified and customized according to the target source code. Conceptual *Solution Snippets* appear

to be the most challenging ones to be adapted for reuse.

If Stack Overflow slightly modifies its Markdown dialect to specify the semantics of code snippets, such as *Solution Snippets*, examples, outputs, and reusability aspects, it would help researchers improve the extraction of datasets of Question-Solution pairs.

There are several threats to the validity of the results of this study. This study heavily relies on human judgment; to minimize possible subjectivity when coding answers, the coders coded batches of answers until the saturation of the categories. Lastly, the results of this study are not generalized over all the questions and answers on Stack Overflow. Because the results of qualitative studies are usually presented with respect to a specific context, the lack of generalizability of this study is not a critical threat to the validity.

7

Conclusions and Future Work

In this chapter, I conclude my study (Section 7.1) and describe the future directions of my study towards creating high-quality datasets of Question-Solution pairs (Section 7.2).

7.1 Conclusions

The two main contributions of my study are: (1) a categorization of how *Solution Snippets* are presented in Stack Overflow's answers and (2) a categorization of how *Solution Snippets* need to be adapted for reuse.

The first categorization has eight categories of how *Solution Snippets* are found in Stack Overflow's answers. After reflecting on these categories, I identified several challenges that researchers who extract Question-Solution pairs from Stack Overflow need to be aware of. First, *Solution Snippets* could be found in code blocks as well as inline with the text; it is required to scan through all the code blocks and inline code snippets in an answer to find *Solution Snippets*. Second, *Solution Snippets* within code blocks are often surrounded by ancillary code (the prologue and epilogue). This ancillary code may contain information that is not relevant to the question but may also contain contextual information about the reusability of *Solution Snippets*. Whenever ancillary code is found within a code block, the *Solution Snippet* and this ancillary code need to be extracted separately. Lastly, a code block may contain multiple *Solution Snippets*; in such cases, it is required to extract each *Solution Snippet* separately.

The second categorization has five different categories of how a *Solution Snippet* could be adapted for reuse. The first two categories (As-is, User-defined functions) are straightforward to reuse, while the last three categories (Placeholders, Customizable variables and/or data, and Conceptual) appear challenging to reuse and require understanding the code to be customized. When a *Solution Snippet* contains customizable variables and/or data, they need to be replaced with actual values and code snippets. When a *Solution Snippet* contains placeholders, they also need to be identified and replaced with suitable code snippets; however, there is no consistency of Stack Overflow users on what to use as placeholders, and thus, it is challenging to identify them. Of all the reusability aspects, Conceptual *Solution Snippets* are the most challenging ones to be reused because such *Solution Snippets* cannot be copied-and-pasted and customized but require writing the entire *Solution Snippet* from scratch.

7.2 Future Work

The results of this study will pave the way for several future works. First, when creating a Question-Solution pair by pairing a title of a Stack Overflow question with the code blocks in it answers, the researchers could tag different semantics of code snippets within those code blocks such as *Solution Snippets*, ancillary code, potential reusability aspects, etc. Subsequently, researchers could leverage such Question-Solution pairs to train machine learning models, such as code generation models.

Second, the manually coded dataset of this study ([6]) could be transformed into a dataset of Question-Solution pairs. Then, it could be used as a manually curated benchmark dataset for source code generation models (for JavaScript, Java and Python) to evaluate such models using an evaluation method (such as Codebleu score [43]).

Lastly, the sample inputs and outputs within code blocks that exemplify the usage of *Solution Snippets* could be used to create test cases to evaluate the quality of the source code generated by machine learning models.

Bibliography

[1] How do i format my posts using markdown or html? `https://stackoverflow.com/help/formatting`. Accessed: 2021-01-08.

[2] Khaldoun M Aldiabat and Carole-Lynne Le Navenec. Data saturation: The mysterious step in grounded theory methodology. *The Qualitative Report*, 23(1):245–261, 2018.

[3] Miltiadis Allamanis and Charles Sutton. Why, when, and what: analyzing stack overflow questions by topic, type, and code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 53–56. IEEE, 2013.

[4] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *International conference on machine learning*, pages 2123–2132. PMLR, 2015.

[5] B Anisah and Nor Afiah MZ. Qualitative research in a nutshell. *International Journal of Public Health and Clinical Sciences*, 1(2):163–166, 2014.

[6] Anonymous. How solution snippets are presented in answers posted on Stack Overflow and how they could be potentially reused. `https://doi.org/10.5281/zenodo.5819318`, January 2022.

[7] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. Sotorrent: reconstructing and analyzing the evolution of stack overflow posts. In Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 319–330. ACM, 2018.

[8] Sebastian Baltes, Christoph Treude, and Stephan Diehl. Sotorrent: Studying the origin, evolution, and usage of stack overflow code snippets. In *2019 IEEE/ACM*

16th International Conference on Mining Software Repositories (MSR), pages 191–194. IEEE, 2019.

[9] Antonio Valerio Miceli Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*, 2017.

[10] Stefanie Beyer, Christian Macho, Massimiliano Di Penta, and Martin Pinzger. Analyzing the relationships between android api classes and their references on stack overflow. *Technical Report*, 2017.

[11] Stefanie Beyer, Christian Macho, Massimiliano Di Penta, and Martin Pinzger. Automatically classifying posts into question categories on stack overflow. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 211–21110. IEEE, 2018.

[12] Stefanie Beyer and Martin Pinzger. A manual categorization of android app development issues on stack overflow. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 531–535. IEEE, 2014.

[13] Eduardo Cunha Campos and Marcelo de Almeida Maia. Automatic categorization of questions from q&a sites. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 641–643, 2014.

[14] Yingkui Cao, Yanzhen Zou, and Bing Xie. Extracting code-relevant description sentences based on structural similarity. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, pages 1–10, 2019.

[15] Kathy Charmaz. *Constructing grounded theory: A practical guide through qualitative analysis*. sage, 2006.

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[17] John W Creswell and J David Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.

[18] JW Creswell. 30 essential skills for the qualitative researcher. 2016.

[19] Michael Crotty and Michael F Crotty. *The foundations of social research: Meaning and perspective in the research process*. Sage, 1998.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[21] Joseph L Fleiss and Jacob Cohen. The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and psychological measurement*, 33(3):613–619, 1973.

[22] Graham Gibbs. *The Sage Qualitative Research Kit, 8 Vols: Analyzing Qualitative Data*. Sage, 2007.

[23] Egon G Guba. The alternative paradigm dialog. *The paradigm dialog*, 1:17–27, 1990.

[24] Egon G Guba, Yvonna S Lincoln, et al. Competing paradigms in qualitative research. *Handbook of qualitative research*, 2(163-194):105, 1994.

[25] J Amos Hatch. *Doing qualitative research in education settings*. Suny Press, 2002.

[26] Rashina Hoda, James Noble, and Stuart Marshall. Developing a grounded theory to explain the practices of self-organizing agile teams. *Empirical Software Engineering*, 17(6):609–639, 2012.

[27] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[28] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

[29] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588*, 2018.

[30] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[31] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. *arXiv preprint arXiv:1802.08979*, 2018.

[32] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[33] Anthony Naaeke, Anastacia Kurylo, Michael Grabowski, David Linton, and Marie L Radford. Insider and outsider perspective in ethnographic research. *Proceedings of the New York State Communication Association*, 2010(1):9, 2011.

[34] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34. IEEE, 2012.

[35] Anh Tuan Nguyen, Peter C Rigby, Thanh Nguyen, Dharani Palani, Mark Karanfil, and Tien N Nguyen. Statistical translation of english texts to api code templates. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 194–205. IEEE, 2018.

[36] Thanh Nguyen, Peter C Rigby, Anh Tuan Nguyen, Mark Karanfil, and Tien N Nguyen. T2api: Synthesizing api code usage templates from english texts with statistical translation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1013–1017, 2016.

[37] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584. IEEE, 2015.

[38] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. An empirical cybersecurity evaluation of github copilot's code contributions. *arXiv preprint arXiv:2108.09293*, 2021.

[39] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack overflow in the ide. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1295–1298. IEEE, 2013.

[40] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Stormed: Stack overflow ready made data. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 474–477. IEEE, 2015.

[41] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering*, 2019.

[42] Musfiqur Rahman, Peter Rigby, Dharani Palani, and Tien Nguyen. Cleaning stackoverflow for machine translation. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 79–83. IEEE, 2019.

[43] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[44] Christoffer Rosen and Emad Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, 21(3):1192–1223, 2016.

[45] David Sikolia, David Biros, Marlys Mason, and Mark Weiser. Trustworthiness of grounded theory methodology research in information systems. 2013.

[46] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. How do programmers ask and answer questions on the web?(nier track). In *Proceedings of the 33rd international conference on software engineering*, pages 804–807, 2011.

[47] Christoph Treude, Martin P Robillard, and Barthélémy Dagenais. Extracting development tasks to navigate software documentation. *IEEE Transactions on Software Engineering*, 41(6):565–581, 2014.

[48] Gias Uddin, Foutse Khomh, and Chanchal K Roy. Mining api usage scenarios from stack overflow. *Information and Software Technology*, 122:106277, 2020.

[49] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, 24(2):637–673, 2019.

[50] Frank F Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. Incorporating external knowledge through pre-training for natural language to code generation. *arXiv preprint arXiv:2004.09015*, 2020.

[51] Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 344–354. IEEE, 2020.

[52] Di Yang, Aftab Hussain, and Cristina Videira Lopes. From query to usable code: an analysis of stack overflow code snippets. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 391–401. IEEE, 2016.

[53] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*, pages 1693–1703, 2018.

[54] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486. IEEE, 2018.

[55] Alexey Zagalsky, Daniel M German, Margaret-Anne Storey, Carlos Gómez Teshima, and Germán Poo-Caamaño. How the r community creates and curates knowledge: an extended study of stack overflow and mailing lists. *Empirical Software Engineering*, 23(2):953–986, 2018.

Appendix A

The questions downloaded from the SOTorrent database

Among the several attributes in the SOTorrent database, I used only the following attributes to filter the questions studied in this research (see Table A.1 for a sample dataset downloaded from SOTorrent).

- **PostTypeId:** This is "1" for questions posted on Stack Overflow and "2" for answers. For the first part of this study, questions were needed, and I filtered posts with PostTypeId equal to "1".
- **Tags:** This usually indicates the programming language related to a question posted on Stack Overflow. For this study, I filtered the questions from the SOTorrent database that have the following three tags: "<JavaScript>," "<Java>," and "<Python>".
- **Score:** This is the number of upvotes minus the number of downvotes for a question or an answer posted on Stack Overflow. I used the score as a proxy for popularity to select questions to study; I ranked the questions in the SOTorrent database by the score in descending order and obtained the top 1000 questions.
- **Id:** This is the identifier that uniquely identifies a question or an answer posted on Stack Overflow. I appended the prefix — "https://stackoverflow.com/q/" — to Ids in order to generate the URLs for the questions used in this study. For example, the Id of the first question in Table A.1 is 679915, and the URL generated from this Id is `https://stackoverflow.com/q/679915`.

- **Title:** This is the title of a question posted on Stack Overflow. The title usually indicates the programming problem.

I used the above attributes, to prepare the Google Sheets for classifying questions in the first part of my study.

Table A.1: A sample dataset used to prepare the Google Sheets for the first part of my study

PostTypeId Id	Tags PostTypeId	Score	Title	Id
1	<javascript>	2730	How do I test for an empty JavaScript object?	679915
1	<javascript>	2328	How can I convert a string to boolean in JavaScript?	263965
1	<javascript>	1941	Open a URL in a new tab (and not a new window)	49078437

Appendix B

How the categories identified in my study evolved

Table B.1 shows the number of answers coded in each iteration, the number of coders who participated in coding, and the categories identified at the end of each iteration (focusing on the two research questions).

The first iteration

The three coders coded ten answers. For RQ1, the following categories were identified: 1. "*Solution Snippet is a one-liner"* (one function call), and 2. "*Solution Snippet contains multiple function calls."* For RQ2, the following categories were identified: 1. *Function declarations*, and 2. *Contain Specific data*.

The second iteration

The three coders coded 43 answers in the second iteration. For RQ1, the categories following the first iteration remained the same. For RQ2, a new category was identified: *"Contain placeholders."*

The third iteration

The three coders coded 23 answers in the third iteration. For RQ1, two new categories were identified: *"Contain examples"* and *"Contain inline Solution Snippets*." The category "*Solution Snippet is a one-liner*" was discarded because the two new

categories identified already included the one-liners. For RQ2, another new category was identified: *"Contain customizable variables."*

The fourth iteration

The three coders coded 26 answers in the fourth iteration. For RQ1, several new categories were identified. For a better understanding of how *Solution Snippets* were presented in the answers, the category *"Contain examples"* was split into two separate categories: *"the code block that contains the Solution Snippet also contains sample input/output,"* and *"the Solution Snippet is an example."* By the end of the fourth iteration, there were eight categories for RQ1 (see Table 4.1 in Chapter 4 for more information). For RQ2, two new categories were identified: *"As-is"* and *"Conceptual."* In addition, two existing categories (*"Contain customizable variables"* and *"Contain Specific data"*) were merged into a single category, *"Contain customizable variables and/or data,"* because most of the *Solution Snippets* that contained customizable variables also contained specific data.

The fifth iteration

The three coders coded 30 answers in this iteration. The categories identified remained unchanged since the previous iteration. The coders believed that the categories had become saturated, and they stopped coding.

The last iteration

To avoid the premature closure of coding in order to ensure the saturation of categories, I continued coding more answers until the top two answers to all of the *how-to* questions were categorized.

Table B.1: Total number of questions and answers studied

Iteration	**No. of answers**	**No. of coders**	**The categories for RQ1**	**The categories for RQ2**
1	10	3	1. *Solution Snippet is a one-liner* (one function call). 2. *Solution Snippet contains multiple function calls.*	1. *Function declarations* (a generic solution) 2. *Contain specific data* (a specific solution)
2	43	3	1. *Solution Snippet is a one-liner* (one function call). 2. *Solution Snippet contains multiple function calls.*	1. *Function declarations* 2. *Contain placeholders* 3. *Contain specific data*
3	23	3	1. *Contain multiple Solution Snippets* 2. *Contain examples* 3. *Contain inline Solution Snippets*	1. *Function declarations* 2. *Contain Placeholders* 3. *Contain customizable variables* 4. *Contain specific data*
4	26	3	See Table 4.1 in Chapter 4 for the categories.	See Table 4.2 in Chapter 4 for the categories.
5	30	3	Remained unchanged.	Remained unchanged.
6	466	1	Remained unchanged.	Remained unchanged.
Total	598	—	—	—

Appendix C

The manually coded dataset of my study

The manually coded dataset of my study is available at [6], which contains two Microsoft Excel files. First, the file `Manual-classification-question-in-SO.xlsx` contains the dataset of questions that I used in my study to identify *how-to* questions (as mentioned in Section 3.3.2) and the classes of those questions (whether a *how-to* question or not). This file contains three sheets, one for each programming language I studied in this research. Each sheet contains the following attributes:

- Language*: The programming language tagged to the question.
- Question's title*: The title of the question.
- Question's URL*: The URL of the question.
- Class: The class of the question — whether it is a *how-to* question or not.

Second, the file `Manual-annotations-Solution-Snippets-in-SO.xlsx` contains the manually coded dataset of the answers to the *how-to* questions (the coders coded at least two top-scored answers to each *how-to* question). This file contains three sheets, each of which contains several attributes of answers for each programming language. These attributes are partitioned into four main sections: the attributes of questions, the general attributes of the answers, the attributes related to the categorization for RQ1, and the attributes related to the categorization for RQ2. Furthermore, this file shows how all the answers studied in my research are categorized into

* Already populated before the coding session.

the saturated categories, as done in the last two iterations mentioned in Section 3.3.3. In the next section, I describe these attributes.

The attributes of questions

- Language*: The programming language.
- *How-to* question's title*: The title of the question.
- *How-to* question's URL*: The URL of the question.

The general attributes of answers

- Answer's URL*: The URL of the answer.
- Number of code blocks: How many code blocks are there in the answer?
- Number of inline code snippets: How many inline code snippets are there in the answer?

The attributes related to the categorization for RQ1

- No code solution or no *Solution Snippet*: Does the answer contain at least one *Solution Snippet*? If the answer simply explain how to solve it in the text without presenting a precise *Solution Snippet*, I considered it as no *Solution Snippet*.
- Number of *Solution Snippets*: How many *Solution Snippets* are there in the answer?
- *Solution Snippet* span over entire code block: How many *Solution Snippets* are there in the answer that span the entire code block?
- A code block with *Solution Snippet* + example input/outputs: How many code blocks are there in the answer that contain *Solution Snippets* as well as sample inputs and outputs?
- A code block with only examples: How many code blocks are there in the answer that contain only sample input/outputs/examples?

* Already populated before the coding session.

- A code block with multiple *Solution Snippets*: How many code blocks are there in the answer that contain multiple *Solution Snippets* within a code block?
- Multi-code block *Solution Snippet*: Does the answer contain at least one *Solution Snippet* that spans over several code blocks?
- Use the code/examples given in the question: Does the answer contain at least one *Solution Snippet* that reuses variables, data or code snippets found in the question's body?
- Library specific *Solution Snippets*: Does the answer contain any library-specific *Solution Snippets*?
- Version Specific *Solution Snippets*: Does the answer contain any programming language version-specific *Solution Snippets*?
- *Solution Snippets* for related questions: Does the answer contain at least one *Solution Snippet* that solves another question that is not explicitly asked in the original question?
- *Solution Snippet* is Inline with text: Does the answer contain at least one *Solution Snippet* that is inline with the text?
- *Solution Snippet* is an API call: Does the answer contain at least one *Solution Snippet* that is only a function call?
- *Solution Snippet* is a library variable: Does the answer contain at least one *Solution Snippet* that is only a variable in a library?
- *Solution Snippet* is a hack: Does the answer contain at least one *Solution Snippet* that is likely to raise a compile/runtime error?
- Contains runnable code snippets (JavaScript-specific): Does the answer runnable code snippets?
- *Solution Snippet* is mixed with REPL (Python-specific): How many code blocks are there in the answer that contain *Solution Snippets* as well as REPL outputs within the code block?

The attributes related to the categorization for RQ2

- As-is: Does the answer contain any *Solution Snippet* that does not need any modifications when reusing?
- UDF: Does the answer contain any User-defined-functions?
- Placeholders: Does the answer contain any *Solution Snippets* containing placeholders, such as print statements, comments, `pass`, etc.?
- Customizable Variable: Does the answer contain any *Solution Snippets* containing customizable variables and/or data?
- Conceptual: Does the answer contain a *Solution Snippet* that exemplifies a programming concept and contains arbitrary function names, variable names, etc.