Master's Thesis

An Empirical Study to Understand Pythonic Lists and Dictionaries Usage in Textbooks

Ruksit Rojpaisarnkit

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

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

Submitted January 29, 2024

Ruksit Rojpaisarnkit

An Empirical Study to Understand Pythonic Lists and Dictionaries Usage in Textbooks*

Ruksit Rojpaisarnkit

Abstract

Coding in a *'Pythonic Way'* involves writing code in a manner that is idiomatic and natural for the Python language, aligning with its philosophy and community conventions. Despite recent efforts to understand various aspects of Pythonic coding, there is still a lack of knowledge regarding how educators transition from non-idiomatic code to Pythonic equivalents. This study focuses on data structures and their introduction in textbooks. I analyze examples of traditional data structures (such as lists and dictionaries) and search for their Pythonic representations. From a dataset of 1,624 examples extracted from 177 curated Python textbooks, I address two research questions. Surprisingly, 69 out of the 177 textbooks lacked any examples of data structures, let alone Pythonic code. The results provide insights into scenarios where using traditional non-Pythonic data structures might still be preferred, considering specific topics (e.g., data science, visualization, statistics), comparison scenarios (readability vs. performance), and other reasons. This work highlights the limited prevalence of Pythonic education and suggests the direction for constructing and maintaining the *Zen of Python* in the next generation of Python programmers by raising the concern about the importance of introducing the Pythonic code when teaching Python programming.

Keywords:

Pythonic, Educational Materials, Teaching

*Master's Thesis, Graduate School of Science and Technology, Nara Institute of Science and Technology, January 29, 2024.

Contents

List of Figures

List of Tables

1 Introduction

Like other programming language communities(e.g. Java, Ruby, JavaScript, etc), the Python community has over time developed its own agreed consensus on how to write Python [3]. From programmer blogs, mailing lists, to official documentation, the term “Pythonic” is used liberally by the Python community. As a result, the community put together the *Zen of Python*, which is a set of 19 design principles [4] for writing computer programs that influence the design of the Python programming language [5]. The principles include concepts such as “Beautiful is better than ugly”, “Explicit is better than implicit”, and “Simple is better than complex”. Intuitively, non-Pythonic code may appear imperfect to an experienced Python developer. [3] showed that the scope of the term “Pythonic” seems to go far beyond concrete source code, referring to a way of thinking about problems and potential solutions.

In recent times, the Python language has grown in popularity due to its ease of use and versatility to be used by professionals and academics outside of computer science. Specifically, Python has been broadly utilized by researchers and practitioners in various fields, including visualization, statistics, data science, machine learning, and AI. Evidence of this popularity is the rise of the usage of Jupyter notebooks [6] and data repositories such as Kaggle,[*] and the emergence of libraries such as Scikit-learn,[†] TensorFlow,[‡], and Keras.[§] With the increase of software artifacts available on platforms like GitHub, Pythonic programmers need to address issues related to size, complexity, and memory limitation. A key factor in Python's popularity might be due to the flexibility that it offers, as Python allows variables and data structures without any data type checking. Re-

[*]`https://www.kaggle.com/`

[†]`https://scikit-learn.org`

[‡]`https://www.tensorflow.org`

[§]`https://keras.io`

cent work now focuses on the various aspects of detecting and utilizing Pythonic practices [7, 8, 9, 10, 11]. For example, [7] devised a catalog of 19 Pythonic idioms and examined their performance and readability. [8] visually revealed that tend to gradually transition over to Pythonic code over time, while [12] showed that writing in Pythonic idioms may save memory and time, and was validated by [13], who found that the impact of Python idioms on code performance varies greatly across idioms. Although there exist works that have explored the usage of Pythonic constructs, it is still unknown the extent to which Pythonic examples appear in teaching resources such as textbooks.

Figure 1.1 and Figure 1.2 are excerpts that show how a textbook introduces the Pythonic method in comparison to the traditional list and dictionary [1, p. 43] and [2, p. 151]. In both examples, I see that the author first presents the traditional data structure, and then proceeds to introduce the Pythonic counterpart. In both cases, the author explicitly states a preference for the Pythonic code over the non-Pythonic approach (traditional list and dictionary).

In this work, I aim to investigate how Pythonic code is taught in a teaching environment. Specifically, I focus on data structures and how they are introduced in textbooks. Before proceeding, I first explain each Pythonic list idiom and demonstrate how each Pythonic code is preferred over the traditional methods.[¶]

1. List comprehension is a concise and expressive technique used to generate new lists by applying a predefined operation to each item within an existing iterable (e.g., a list, tuple, or range), enhancing code readability and minimizing the necessity for explicit looping constructs.

```
# Note: n is integer
result_list = [el for el in range(n)]
```

2. Dict comprehension is an efficient approach to creating dictionaries by specifying key-value pairs. Dict comprehension provides an elegant and Pythonic way to accomplish such tasks with efficiency.

```
# Note: n is integer
dict_compr = {k: k*2 for k in range(n)}
```

[¶]These examples were taken from [12]

3. collections.defaultdict. is a specialized type of dictionary in Python that enables you to set a default value for keys that do not exist in the dictionary. It not only adjusts the code but also enhances the readability.

```python
# Note: n is integer
#        n_list is list of n integer elements
from collections import defaultdict
def add_new_value_in_dict_pythonic(n_list):
    num_dict = defaultdict(int)
    for key in n_list:
        num_dict[key] -= key
    return num_dict
```

4. collections.deque as an abbreviation for "double-ended queue." This data structure is versatile and can be used as a queue, stack, or for manipulating sequences effectively. Its flexibility makes it a valuable choice for implementing algorithms that require both stack and queue functionalities. Additionally, it offers advantages in terms of memory efficiency and execution speed.

```python
from collections import deque
def add_new_value_in_queue_pythonic(n):
    num_queue = deque([])
    for num in range(n):
        if num % 4 == 0:
            num_queue.append(num)
        elif num % 4 == 1:
            num_queue.appendleft(num)
        elif num % 4 == 2:
            num_queue.pop()
        else:
            num_queue.popleft()
    return num_queue
```

As a concrete example, a list of the squares of the numbers from 1 to n, that is $[1, 4, 9, 16, 25, \ldots, n^2]$, can be created by traditional means as follows:

```
squares = []
for k in range(1,n+1):
   squares.append(k * k)
```

With list comprehension, this logic is expressed as follows:

```
squares = [k * k for k in range(1,n+1)]
```

Figure 1.1: Example of a traditional list to Pythonic list comprehension [1, p. 43]

Similarly, given a dictionary d, you could create a shallow copy named d1 by coding out a loop:

```
>>> d1 = { }
>>> for somekey in d:
...    d1[somekey] = d[somekey]
```

or more concisely by

```
>>> d1 = { }; d1.update(d).
```

However, again, such coding is a waste of time and effort and produces nothing but obfuscated fatter, and slower code. Use

```
>>> d1=dict(d)
```

Figure 1.2: Example of a traditional dictionary to Pythonic dict comprehension [2, p. 151]

1.1 Research Statement

- **(RQ1) What are the kinds of textbooks introduce Pythonic data structures?**
  Motivation: To explore which textbooks are more likely to teach the Pythonic one-liners to initialize and populate lists and dictionaries. Answering this research question will identify which target audience makes use of this Pythonic coding style.

- **(RQ2) What are the case-scenarios where a textbook prefers to use traditional data structures?**
  Motivation: There may be cases where using the traditional data structure is preferred over the Pythonic counterparts. Hence, answering this research question will allow us to understand why the knowledge of traditional data structures and algorithms is important for programmers.

1.2 Replication package

To facilitate replication and future work in the area, I have prepared a replication package, which includes raw data, the manually labeled dataset, and the scripts for reproducing my analysis. The package is available online at `https://zenodo.org/doi/10.5281/zenodo.10053630`.

1.3 Organization

The remainder of the thesis is organized as follows. Section 2 illustrates the background and related work of this study. Section 3 describes how I collected textbooks and extracted code examples. Sections 4 and 5 show the experiments that I conducted to address RQ1–2, respectively. Section 6 discusses the recommendations for practitioners, researchers, and educators based on my study results. Section 7 discusses the threats to validity. Section 8 draws the conclusions of my study and highlights opportunities for future work.

2 Background and Related work

In this section, I describe the detail about background and related work of this research including Pythonic coding and Mining Programming textbooks.

2.1 Pythonic Coding

Python is well-known for its Pythonic idioms, which represent notable programming styles and features of a language [10]. Several studies have explored the patterns of Python idioms by searching for Python textbooks or online resources. For example, Alexandru et al. [7] devised a catalog of 19 Pythonic idioms collected from several textbooks and manually examined the performance and readability of the identified idioms. Farooq and Zaytsev [3] further reinforced the statements made in the original paper of Alexandru et al. by generalizing them to a wider collection of idioms. Meanwhile, some works investigated the role of Pythonic idioms in open-source projects and their performance. Sakulniwat et al. [8] visually revealed that developers tend to adopt the open idiom over time. The preliminary experiments by Leelaprute et al. [12] showed that writing in Pythonic idioms may save memory and time. Zhang et al. [13] studied 24,126 pairs of nonidiomatic and functionally-equivalent idiomatic code for the nine unique Python idioms and found that the impact of Python idioms on code performance varies greatly across idioms.

Some automated tools have been developed to support Pythonic coding. Zhang et al. [14] designed and implemented an automatic refactoring tool to make Python code idiomatic with nine idiom types, demonstrating its practicality and usefulness. Phan-Udom et al. [9] introduced an automated tool named Teddy to assist in checking idiom usage. Evaluation results suggested that Teddy has a high precision in detecting idiomatic and non-idiomatic Python code.

The implication of these related works shows the importance of Pythonic idioms in terms of various benefits. For instance, using Pythonic idioms improves the code and execution performance, saving memory usage, etc. Moreover, there is an effort to develop the tool to support Pythonic coding by refactoring and converting the traditional code to Pythonic code.

2.2 Mining Programming Textbooks

Various approaches are employed to gain insights in software engineering research, such as conducting surveys, interviewing relevant individuals, and mining code or useful information from software repositories. Despite these methods, there has been limited exploration of mining textbooks in the context of software engineering research.

Textbooks are widely recognized as a valuable resource for teaching introductory programming languages [15, 16]. They not only provide a wealth of example programs but also serve as a reference for solving specific problems [17]. Numerous studies have examined the role of textbooks in education, specifically focusing on various programming languages. Börstler et al. [18] studied in which experienced educators evaluated the quality of object-oriented example programs for novices from popular Java textbooks and they suggested that educators should be careful when taking examples straight out of a textbook. Mazumder et al. [19] reported that none of the 15 commonly used introductory Java textbooks provide explanative diagrams for variables, arrays, and objects. Almansoori et al. [20] analyzed the discussion of security topics and the use of unsafe functions in the thirteen textbooks used in the top 30 R1 universities in the US. Their results indicate that many textbooks fail to provide warnings about the use of unsafe functions or teach students how to use them safely.

At the same time, several attempts have been made to utilize external components in order to enhance the effectiveness of textbooks. For example, Alpizar-Chacon et al. [21] introduced an ontology-based approach that integrates textbooks with smart interactive content (such as worked examples and code animations) for learning programming. Huang et al. [22] proposed a framework called EMRCM, which constructs multiple relationship knowledge graphs of concepts from multiple sources, including textbooks. Wu et al. [23] presented a programming language learning service that utilizes Stack Overflow posts as pragmatic knowledge of programming languages, addressing the lack of pragmatic knowledge in textbooks.

Briefly, while various methodologies such as surveys, interviews, and code mining are commonly employed in software engineering research, there has been a notable gap in exploring the rich resources of textbooks within this domain. Despite textbooks being widely acknowledged for their value in teaching programming languages, existing studies highlight shortcomings in the quality and coverage of example programs, as well as the inadequate treatment of crucial topics like security. Notably, the focus on textbooks has been fragmented, with specific attention given to individual programming languages. On another hand, efforts to augment traditional textbooks with external components, such as smart interactive content and knowledge graphs, showcase endeavors to enhance their educational efficacy. Moving forward, a more comprehensive exploration of textbooks, coupled with innovative approaches to supplement their content, could significantly contribute to the advancement of software engineering research and education.

3 Methodology

In this section, I describe the overview and how the data was collected for the empirical study, including the collection of Python textbooks and the extraction of code examples. Figure3.1 shows the overview of this study.

3.1 Collecting Python Textbooks

In order to collect the dataset for the experiment, I sourced Python textbooks from three different sources in order to mitigate the risk of insufficient data.

From Related Work The first is from the dataset made available by Robles et al. [24], consisting of 80 textbooks. They collect Python-related textbooks from their university library. This set of textbooks was also used by Alexandru et al. [7].

From GitHub Repositories The second dataset was constructed from free Python books that were made available via GitHub repositories. To search for these, I utilized the GitHub topics that used the tag `python-book`.[*] I successfully parsed through the 20 repositories and collected a list of 76 distinct textbooks.

From Amazon Reviews The final data source is from Amazon reviews. To compile the list, I searched on Amazon.co.jp[†] using the keyword ‘python book’ and sorted the books by customer review rating. I collected the top 100 Python textbooks based on the review ratings in ascending order. Afterward, I performed

[*] `https://github.com/topics/python-book`

[†] `https://www.amazon.co.jp/s?k=python+book`

Table 3.1: Collected Dataset of Textbooks

Collected Textbooks	
# from Related Work	80 textbooks
# from GitHub Repositories	76 textbooks
# from Amazon Reviews	100 textbooks
# Python Books downloaded	21 textbooks
— Python Books not downloadable	79 textbooks
Total	177 textbooks

a Google search to determine if there were any free or open versions of these textbooks available for download. As a result, 21 of them could be downloaded.

As shown in Table 3.1, I collected a total of 177 textbooks that were published between 2001 and 2023 for the following analysis, which combined the data from different sources and removed any duplicate textbooks. Note that each textbook is written in English, and I converted all of the different formats of textbooks consistently into PDF format by using the PDF library.

3.2 Extracting Code Examples

To extract the code examples from textbooks, I applied regular expressions to identify and extract the different Pythonic and non-Pythonic code examples from the 177 textbooks. I use the PDFminer[‡] library to extract texts from textbooks and regular expression to extract the code examples from texts. I construct a total of six regular expressions for the extracting process. The first two expressions are to identify the traditional data structures:

1. *Traditional list (TradLists)* - the search string includes initializing a list (x = []), and then appends elements to the list using the `append` function.

[‡]`https://pypi.org/project/pdfminer/`

2. *Traditional dict (TradDicts)* - the search string includes initializing a dictionary (x = {}), followed by assigning a value to each key (e.g. x[key] = value).

Next, I introduce the four regular expressions to identify Pythonic data structures as follows:

1. *List comprehension (ListComp)* - the search string includes initializing a list using a one-line for loop to append elements into the list.

2. *Dict comprehension (DictComp)* - the search string includes initializing a dictionary using a one-line for loop to assign the value to the key.

3. *Collections.deque (Deque)* - the search string includes the initialization of the collections.deque(), which is a list-like data structure.

4. *Collections.defaultdict (Defaultdict)* - the search string includes the initialization of the collections.defaultdict(), which is a dictionary-like object.

Table 3.2 shows the syntax of the regular expression used for extracting the code example from textbooks. Table 3.3 shows the statistical summary of the extracted code examples. As a result of regular expression extraction, a total of 1,624 examples were identified, with 1,231 classified as Pythonic and 393 classified as non-Pythonic. For example, out of the Pythonic examples, 849 are related to List comprehension and 157 are related to Dict comprehension.

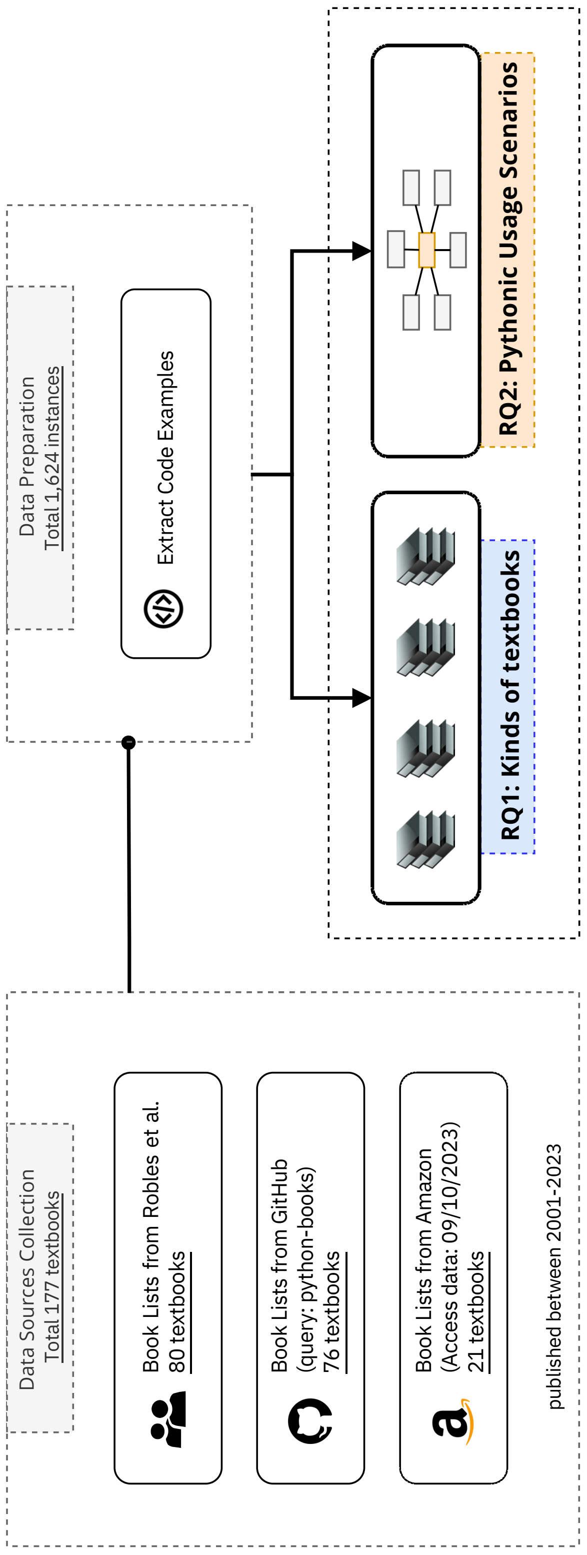


Figure 3.1: Study Overview

Table 3.2: Regular expressions used to identify instances in Python textbooks

No.	Python Data Structure	Regular Expression
1	Traditional list	(\w+\s*=\s*\[.*\]\s*\n*.*\s*for.*in.*:\s*\n*.*\.append\(.*\))
2	Traditional dict	(\w+\s*=\s*\{.*\}\s*\n*.*\s*for.*?in.*?:.*?\n\s*.*?\[\w+\]\s*=\s*[\w*]+\d*)
3	List comprehension	(\w+.*\s*=\s*\[\s*\w+.*\s+for\s+\w+.*\s+in\s+\w+.*\])
4	Dict comprehension	(.*\{\s*.*\s*:\s*.*\sfor\s.*\})
5	Collections.deque	(deque\(.*?\))
6	Collections.defaultdict	(defaultdict\(.*?\))

Table 3.3: Extracted Code Examples

Extracted code examples	
# **Pythonic**	
List comprehension	849 examples
Dict comprehension	157 examples
Collections.defaultdict	85 examples
Collections.deque	140 examples
Total	1,231 examples
# **non-Pythonic**	
Traditional list	220 examples
Traditional dict	173 examples
Total	393 examples
Total	1,624 examples

4 Kinds of textbooks (RQ1)

In this research question, I focus on the characteristics of textbooks that introduce Pythonic data structure and how textbooks present Pythonic data structures. Understanding data structures is essential for coding, but there is no evidence that it needs to be included in every type of programming book. For example, in what kinds of textbooks do they teach concepts like Dict comprehension and List comprehension. Furthermore, a crucial aspect of the investigation involves identifying textbooks that refrain from introducing Pythonic data structures and clarifying the reasons behind it in the next research question.

4.1 Approach

To address the first research question, I analyzed each textbook based on its different characteristics. In this work, I make the assumption that the title of a book is the strongest indicator is whether a textbook will be attractive to its audience, hence I investigated the textbooks according to their titles. To do so, I performed the card sorting approach similar to related work from Zimmermann [25] to manually analyze the titles of textbooks. Card sorting is a commonly used technique in software engineering useful to derive taxonomies from data. The card sorting was performed by myself and my colleague starting with analyzing the textbook title for the first iteration using *open card sorting* where the characteristics were not predefined and adapted during the sorting process. Then I used the initialized taxonomy from the first iteration to employ *closed card sorting.* I sorted the characteristics of each iteration according to the previous one until the *closed card sorting* succeeded. Afterward, the generated taxonomy was validated by another colleague, who possesses extensive research experience in the field of software engineering, as well as ten years of teaching experience in Python. In the end, I classified the 177 textbooks into the following four groupings:

1. *A reference, guide or cookbook (Reference).* The first grouping is based on identifying textbooks that are reference textbooks that might also be useful for advanced readers who already have experience in programming. Examples include textbooks that either have a terms guide, reference, or cookbook in the title. I assume that these textbooks should contain most Pythonic elements as references.

2. *Focused on a particular topic or theme (Topical).* The second grouping is related to textbooks that target a particular topic. For instance, the topic might be data science, machine learning, or finance. Our assumption is that these

3. *Focused on a particular Python technology (Library).* The third grouping includes any textbook that explicitly mentions Python technology in the title. Examples include the libraries Pandas, Numpy, or the frameworks such as Django.

4. *Generic textbook on Python (Generic).* The final grouping consists of generic titles that do not fit into any of the other categories. Examples include introduction and beginner textbooks on Python.

During the analysis, I found that there were some textbooks that could fit into multiple groupings. To mitigate this threat, I intentionally performed the classification in a specific order of classification (*Reference*→ *Topical*→ *Library*→ *Generic*).

After classifying the kinds of textbooks, I took 1,624 extracted code examples from Section II to determine whether a given textbook includes Pythonic data structure or not. This further leads to the segregation of the textbooks into three distinct categories of Pythonic usage, as follows:

- *Contain Pythonic* - refer to the textbook that contains at least one Pythonic code example in their content.

- *No Pythonic* - refer to the textbook that does not contain any Pythonic data structure

- *No Python data structure* - refer to the textbook that does not contain any Python data structure.

Based on this classification, I calculate the relative proportions of each type of Pythonic usage within each grouping of textbooks.

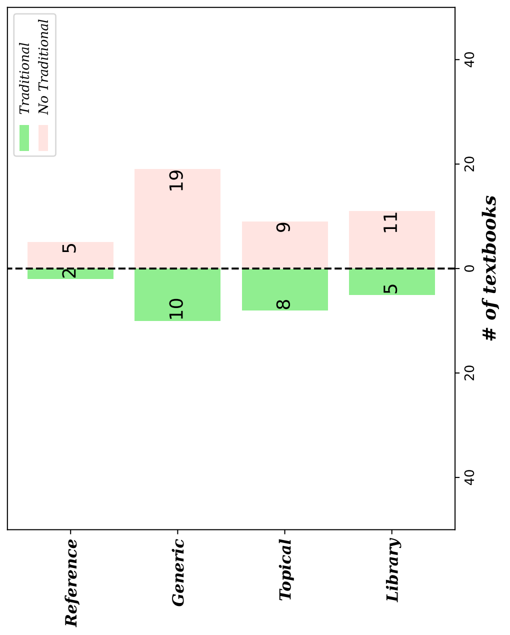
4.2 Results

Figure 4.1 presents the result of the RQ1 analysis. There are four observations from the result as follows:

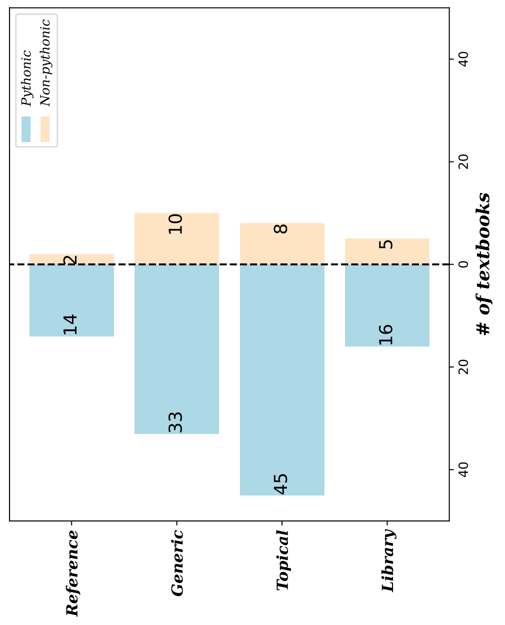
Observation 1: 108 out of 177 textbooks (61.01%) contain Pythonic data structures. The results indicate that out of the 108 textbooks analyzed, all of them incorporate Pythonic practices into their content, suggesting a strong tendency within this category. However, there are 25 textbooks that do not include any Pythonic practices. This means that while these textbooks introduce the concepts of data structure, they do not teach the Pythonic version of data structure.

Observation 2: The majority of the textbooks that include Pythonic data structures are from Topical. Among the 108 textbooks that contain Pythonic data structures, the majority, 45 textbooks, were classified into category Topical, followed by 33 textbooks classified into Library, 16 textbooks classified into Generic, and 14 textbooks classified into Reference. On the other hand, among the textbooks that do not contain Pythonic data structures, the largest proportion was found in category Library, with 10 textbooks, followed by category Topical, with 8 textbooks.

Observation 3: There are 44 textbooks (24.86% of all textbooks) that do not introduce any data structure in textbook From another point of view, in Figure 4.1 - b, there are 44 textbooks remaining do not introduce any Python data structure. This means that these textbooks do not cover any concepts related to data structures. This finding prompted me to conduct a more in-depth analysis to determine the reasons behind the absence of Python data structure in these textbooks.



(b) Traditional vs. No Traditional



(a) Pythonic vs. non-Pythonic

Figure 4.1: Answering RQ1, the proportion of textbooks that (a) contain Pythonic data structures (b) do not contain Pythonic data structures. Furthermore, for (b), identifying textbooks that do not use data structures (List and Dictionary) at all.

Observation 4: Most of the textbooks that lack Pythonic data structures are from Generic In addition to the previous observation, the results show that the majority of textbooks that do not introduce any data structures in their content are generic textbooks. This leads us to examine examples of textbooks that do not cover the concept of data structures, which is a common concept in programming. Some examples of such textbooks include “Beginning Django CMS”, “Python for Google App Engine”, and “Python XML”.

RQ1 Summary:
the result of RQ1 sheds light on the varied landscape of Python data structure coverage in textbooks, highlighting both the widespread adoption of Pythonic practices and the existence of significant gaps in certain categories and generic textbooks. This information lays the groundwork for further investigations and improvements in Python education resources.

5 Pythonic Data Structure Usage Scenarios (RQ2)

In this research question, I focus on a detailed exploration of when textbooks choose to introduce Pythonic data structures versus sticking with traditional ones. Despite the acknowledged advantages of Pythonic idioms highlighted in Section 2 wherein the Pythonic idioms construct similarly to traditional construct, there remains a crucial question: why do some textbooks still opt for traditional data structures instead of embracing the Pythonic versions? This research question seeks to clarify the rationale behind such cases, aiming to provide a comprehensive understanding of the factors influencing instructional choices in this context.

5.1 Approach

In addition to the exploration in RQ1, which focused on uncovering the types of textbooks that did and did not introduce the Pythonic data structure, it is worth noting that 48 textbooks (representing 27.12%) with 167 examples continued to use the traditional version even after introducing the concept of the Pythonic data structure. This led to the formulation of the following research question, where I aimed to determine the reasons behind this persistent preference for the traditional data structure format.

Similar to the previous work by Cruzes and Dyba [26], I applied the thematic analysis to identify and select textbooks wherein the sequence indicated a persistent use of traditional data structures even after the introduction of Pythonic alternatives. Specifically, the thematic analysis involved the four following sequential steps:

1. *Manual inspection of textbook pages*: I began by scrutinizing the pages in these textbooks where examples of traditional data structure usage were found.

2. *Case scenario initialization*: Next, I initiated individual cases for each code example discovered during the inspection.

3. *Labeling of each case and classification*: I then systematically each case to identify opportunities for merging them based on common themes or patterns.

4. *Finalization and Taxonomy Definition*: Finally, I arrived at a conclusive set of themes and patterns and defined a clear taxonomy based on our findings.

These steps were carried out collaboratively by myself and my colleague during the initial round of analysis. Following that, a second round of review was conducted by another colleague, who has over ten years of research and teaching experience.

As a result of our analysis, I have refined the taxonomy into four distinct groups, as outlined below:

- *topic-scenario* - pertains to examples where the author employs the traditional version of code in specific scenarios that involve particular modules or libraries. For example, it may involve the use of non-Pythonic lists in the context of Natural Language Processing, Plot Visualization (using libraries such as Matplotlib), or Regression Models. For example, in the book "Python Data Analytics" [27, p. 283], the author utilizes Traditional list with NumPy library to calculate the distribution of wind speed. Also in Figure 5.2, showed the example of use cases of the sub scenario called Data science from the book [28, p. 207].

- *comparision-scenario* - typically involves evaluating the performance and readability between the traditional code and a refactored version. Encompassing situations where the author references traditional code as a point of reference to introduce more advanced code or other related libraries and functions, such as filter() and map(). For instance, in the book "Fluent

Python" [29, p. 21], the author demonstrates a comparison between the readability of Traditional list and List comprehension. Additionally, on page 38 [29, p. 38] of the same book, an example of code snippet comparison is shown, showcasing a comparison between List comprehension code and Pythonic coding style.

- other-scenarios - refer to cases where the author provides code examples or exercises but employs traditional code without offering an explanation or any specific references to other concepts or technologies. For example, in "Python Playground" book [30, p. 113], the Traditional list function is mentioned as part of a code example in a function, but the specific context of its usage is not detailed.

5.2 Results

As a result of a qualitative analysis, Figure 5.1 shows the use case themes for which traditional data structures are still necessary, even after the introduction of Pythonic data structures. Figures 5.2, 5.3, and 5.4 illustrate the relevant examples in terms of three use cases. There are three observations from the result as follows:

Observation 1: The majority of the scenario where Traditional list were used are topic-scenario. Based on our thematic analysis, I found a total of 88 Traditional list code examples that appear after the introduction of Pythonic data structures. There are 39 code examples within *topic-scenario*, making up 44.32% of the total. 36 code examples are categorized under *comparision-scenario*, accounting for 40.9% of the total. The remaining 13 code examples fall into the *other-scenarios*, representing 14.77% of the total. These findings suggest that there is a continued need for Traditional lists within specific scenarios, particularly in the *topic-scenario* even after the introduction of Pythonic data structures.

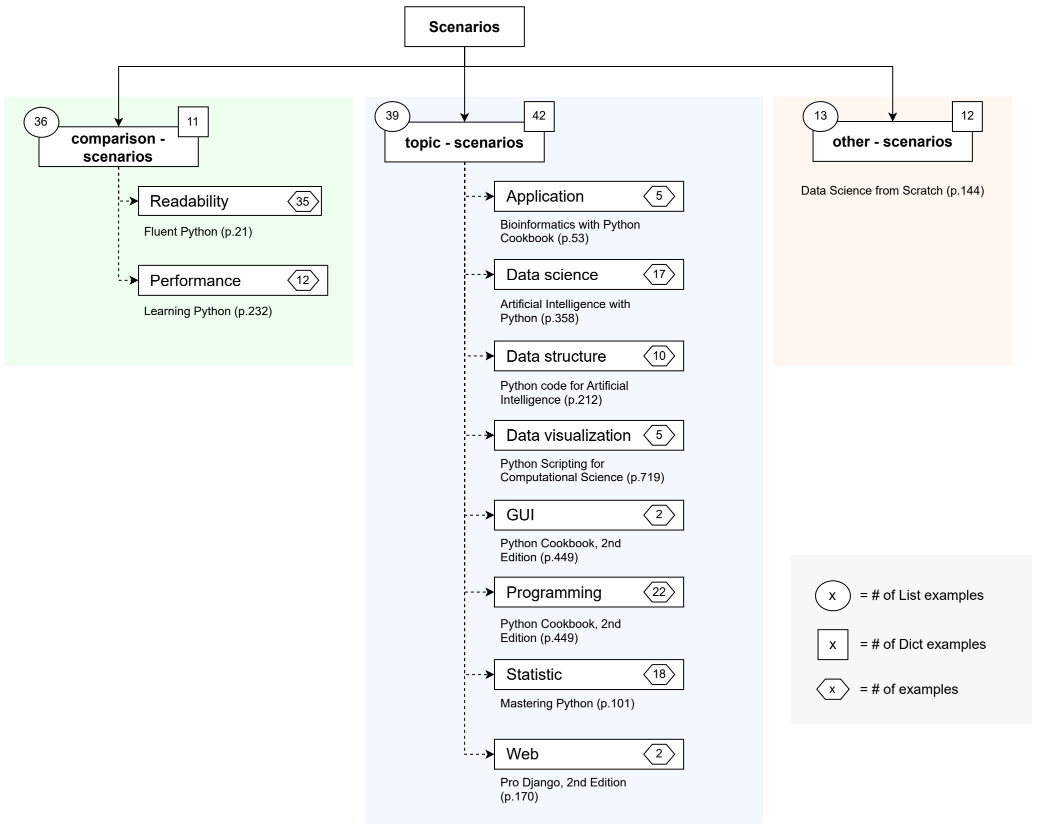


Figure 5.1: Example of Scenarios that still use Traditional list and Traditional dict.

Observation 2: The majority of the scenario where Traditional dict were used are also topic-scenario. Similar to previous observations regarding Traditional list, the result reveals that there are 65 Traditional dict code examples that appear after the introduction of Pythonic data structure. There are 42 code examples within *topic-scenario*, making up 64.62% of the total. 11 code examples are categorized under *comparision-scenario*, accounting for 16.92% of the total. The remaining 12 code examples fall into the *other-scenarios*, representing 18.46% of the total. Consistent with the findings related to Traditional list, these results also suggest that there is an ongoing need for Traditional dictwithin specific scenarios, particularly in the *topic-scenario* even after the introduction of Pythonic data structures.

Observation 3: The example scenarios of topic-scenario are Data Science, Data Visualization, and GUI. As a combination of Traditional list scenarios and Traditional dict scenarios, the topic-scenario is the most popular scenario. Our result reveals that the topic-scenario consists of 8 sub scenarios including Application, Data Science, Data Structure, Data Visualization, GUI, Programming, Statistics, and Web. For example, programming, in the book "Python Cookbook, 2nd Edition"

RQ2 Summary: The result suggests that, despite the introduction of Pythonic data structures, Traditional list and Traditional dict remain essential in specific scenarios, particularly within the topic-scenario. The examples provided offer context and present instances where the use of traditional data structures is still prevalent, contributing to a nuanced understanding of their ongoing relevance in certain Python programming contexts.

topic-scenario

The `_grid` is created using a list comprehension inside a list comprehension. Using list replication such as $[[char] * columns] * rows$ will not work because the inner list will be shared (shallow-copied). I could have used nested for in loops instead:

```
    _grid = []
    for row in range(_max_rows):  % Note: corrected "For" to "
   for"
        _grid.append([])
        for column in range(_max_columns):  % Note: corrected "
   _mas_columns" to "_max_columns"
            _grid[-1].append(_background_char)
```

This code is arguably trickier to understand than the list comprehension and is much longer.

Figure 5.2: Example of topic-scenarios

comparison-scenario

I will write a small piece of code that collects the results of divmod(a, b) for a certain set of integer pairs (a, b).

```python
for a in range(1, mx):
    for b in range(a, mx):
        dmloop.append(divmod(a, b))
print('for loop: {:.4f} s'.format(time() - t)) # elapsed time
t = time()  # start time for the list comprehension
dmlist = [
    divmod(a, b) for a in range(1, mx) for b in range(a, mx)]
print('list comprehension: {:.4f} s'.format(time() - t))
```

As you can see, we are creating three lists: dmloop, dmlist, dmgen (divmod-for loop, divmod-list comprehension). We start with the slowest option, the for loops.

Figure 5.3: Example of comparison-scenarios

other-scenario

The simplest way to do this is to split your data set, so that (for example) two-thirds of it is used to train the model, after which we measure the model's performance on the remaining third.

```
def split_data(data, prob):
    """split data into fractions [prob, 1 - prob]"""
    results = [], []
    for row in data:
        results[0 if random.random() < prob else 1].append(
row)
    return results
```

Often, we will have a matrix x of input variables and a vector y of output variables.

Figure 5.4: Example of other-scenarios

6 Implications

In this section, I list how the results have implications for practitioners, researchers, and educators of not only Python but programming in general.

6.1 Not all textbooks contain Pythonic examples

Especially for students who wish to learn Python, the results (RQ1) show that Pythonic code is not always taught in textbooks. I do understand that the size of the dataset might also cause some concern and the quality of the textbooks. However, this in itself also brings the question of what is the characteristic of a good Python textbook. And should fundamental concepts like lists and dictionaries be mandatory. These are interesting research directions and would open up opportunities for researchers and students alike.

Another reason why textbooks might not contain Pythonic code is that the textbook is not targeted toward an audience that would not necessarily be required to learn how to code. This also brings up an interesting topic of whether or not fundamental programming concepts are required when learning a programming language, especially with respect to its future evolution and maintenance. This should open up new research avenues. Another interesting future direction would be to survey members of the Python community on their thoughts on how Pythonic data structures are not taught in textbooks. Whether or not this is a requirement only for the Open Source Python community, or whether it should be implemented for all Python programmers.

6.2 Considerations for Non-Pythonic Data Structures

The key results from RQ2 indicate that the traditional way to initialize and use lists and dictionaries is still valid. There are cases where using a non-Pythonic data structure is still feasible, and there are cases where Pythonic cannot be implemented. Although there is recent work that indicates automated means to refactor for more Pythonic code, and its performance benefits are valid, there is a need to take into account the situation where the Traditional list and Traditional dict are being integrated since in some situations, the List comprehension and Dict comprehension cannot be used.

6.3 Automation may not be trivial

Related to Implication 2, due to the different contexts by which Pythonic code cannot be applied, tool support is not as trivial. Therefore, researchers and practitioners will need to understand the different contexts in which Pythonic refactoring can occur.

For instance, results from RQ2 show that there are several scenarios where it might be preferred to still use the traditional way to use lists and dictionaries. Specifically, the technology limitations, like the usage of a specific library might dictate why a Pythonic code cannot be used. Understanding these cases, is useful, and might spark future tool support on how to overcome these barriers, like fixing these libraries.

6.4 Pythonic Coding Learning

RQ2 results suggest that Pythonic coding does not consistently follow a predefined sequence. While textbooks may not always cover the complete set of Pythonic idioms for creating lists and dictionaries, an interesting observation is that authors tend to align with Pythonic styles when utilizing comprehensions. This lack of a clear progression in Pythonic learning sequences raises questions about how educators should approach teaching Pythonic coding, prompting further exploration into effective pedagogical methods and sequencing for imparting Pythonic practices.

7 Threats To Validity

In this section, I discuss the threats to the validity of this study, which are the external, construct, and internal threats to validity.

7.1 External Threats

External threats are with regard to the ability to generalize based on the results, especially since I use a curated list of Python textbooks. Two potential threats are summarized. The first threat arises from the fact that the experiment is evaluated based on Python language. Therefore, the observations may vary across different programming languages (e.g., Java and C). However, Python is now widely regarded as the dominant language in various fields, including education and software development. I believe its significance as a representative language is worth exploring. Since the research is specific to Python Idiomatic code, it is only natural to focus on Python language. The future work would also explore other idiomatic codes.

The second threat would occur during the selection of the textbooks. Analyzing a total of 177 textbooks might not capture the breadth and diversity of the entire collection. Nevertheless, I detected 1,624 examples of Pythonic List usage. Having substantiated the results from the studied textbooks, I believe the findings have significant implications for practitioners, researchers, and educators. Furthermore, I open and present all the textbooks for the reader and also for replication of the dataset. Future work may include expanding the datasets, as well as exploring other teaching materials such as online blogs, tutorials, and teaching curriculums.

In future work, I plan to extend the framework to support additional programming languages. I also aim to include a diverse range of Pythonic codes. In this study, I only investigate four Pythonic codes that are only related to data structures. However, as the initial study, I believe that these insights are crucial and serve as a foundation for further research on the systematic utilization of information in textbooks to support educators and learners.

7.2 Construct Threats

Construct threats refer to the degree to which the measurements capture what I aim to study. A potential threat may arise in the accuracy of extracting the code examples. There are obvious limitations to the automated approach. To migrate this threat, I manually inspected all extracted examples of the examples, especially for RQ2. Therefore, I believe that the constructed dataset is sufficiently valuable to be mined and provide insightful results. In order to create a much larger scale study, an automated method might be required with a larger accuracy. This is seen as potential future work.

7.3 Internal Threats

Internal threats denote the approximate truth about inferences regarding cause-effect or causal relationships with the results. To address the proposed research questions, I performed a series of qualitative analyses and grouping of the titles. This classification was derived during the study, hence, there is a threat that the title of the textbooks might be misleading, or that there are biases in the classification, especially due to the subjective nature of the classification.

To relieve such a threat, I carefully conducted the manual coding in a systematic manner with multiple authors including experienced educators. I performed a round-table iteration with multiple rounds. I am confident that due to the large sample size, the groupings are valid.

8 Conclusion and Future Outlook

In this work, I focus on Pythonic usage of data structures (i.e., lists and dictionaries) and investigate how they are introduced in textbooks. I mined 1,624 code examples from 177 Python textbooks. Specifically, I examined two aspects: (i) the kinds of textbooks that introduce Pythonic data structures, and (ii) the scenarios in which a textbook prefers to use traditional data structures. Results show that 69 out of the 177 textbooks did not include any example of data structures, let alone the Pythonic code to write this. This work clearly shows that Pythonic education is not as prevalent, and opens up new directions on how to maintain this *Zen of Python* for the next generation of Python programmers. The potential future directions include (i) surveying members of the Python community on their perceptions of how Pythonic data structures are not taught in textbooks, (ii) exploring the characteristics of other Pythonic idioms, and (iii) investigating strategies for writing an better Python textbook.

Acknowledgement

I am profoundly grateful to the following individuals, whose unwavering support and encouragement have been instrumental in the successful completion of my Master's degree and this thesis.

Foremost, my heartfelt thanks go to my thesis advisor, Professor Kenichi Matsumoto. His invaluable guidance and the opportunity to join the Software Engineering Laboratory have been pivotal to my academic journey. Professor Matsumoto's continuous support and encouragement throughout my master's program have been a source of inspiration.

I extend deep appreciation to my esteemed co-supervisor, Professor Raula Gaikovina Kula. His expertise and guidance have significantly shaped my research journey. Profound gratitude is owed for sharing his extensive knowledge and offering invaluable lessons on research conduct and precise paper writing. The thoughtful feedback provided by Professor Raula Gaikovina Kula has played a crucial role in advancing my research.

Special acknowledgment is due to the members of my thesis committee, Professor Keiichi Yasumoto, Professor Takashi Ishio, and Professor Kazumasa Shimari. Their insightful comments and thoughtful suggestions have significantly contributed to refining the quality and depth of my thesis.

I am indebted to Professors Kenichi Matsumoto and Raula Gaikovina Kula for the remarkable opportunity they provided during my Master's degree. Participating in an international conference and engaging in an enriching exchange program at King Juan Carlos University, Spain broadened my horizons significantly.

My gratitude extends to Professor Gregorio Robles for his invaluable support and guidance during my lab stay at King Juan Carlos University, Spain. His dedication and expertise greatly contributed to the success of my research.

Moreover, I also want to express my appreciation to my friends and lab mates, whose camaraderie and support made this journey more enjoyable. Overcoming research challenges together has been a testament to the strength of our collaborative spirit.

Lastly, I am deeply thankful to my family for providing me with the opportunity to study abroad. Their unwavering support and hope have been my pillars of strength throughout this academic endeavor.

Bibliography

[1] M. T. Goodrich, M. H. Goldwasser, and R. Tamassia, *Data Structures and Algorithms in Python 1st Edition.* Wiley; 1st edition, 2013.

[2] A. Martelli, A. Ravenscroft, and D. Ascher, *Python Cookbook Second Edition.* O'Reilly Media, Second edition, 2005.

[3] A. Farooq and V. Zaytsev, "There is more than one way to zen your python," in *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, 2021, pp. 68–82.

[4] T. Peters, "PEP 20 – The Zen of Python | Python.org," https://www.python.org/dev/peps/pep-0020/, aug 2004, (Accessed on 11/16/2021).

[5] "Glossary ,Äî python 3.10.0 documentation," https://docs.python.org/3/glossary.html#Pythonic, Nov. 2021, (Accessed on 11/17/2021).

[6] L. Quaranta, F. Calefato, and F. Lanubile, "Kgtorrent: A dataset of python jupyter notebooks from kaggle," feb 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4468523

[7] C. V. Alexandru, J. J. Merchante, S. Panichella, S. Proksch, H. C. Gall, and G. Robles, "On the usage of pythonic idioms," in *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2018, 2018, p. 1,Äì11.

[8] T. Sakulniwat, R. G. Kula, C. Ragkhitwetsagul, M. Choetkiertikul, T. Sunetnanta, D. Wang, T. Ishio, and K. Matsumoto, "Visualizing the usage of pythonic idioms over time: A case study of the with open idiom," in *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP).* IEEE, 2019, pp. 43–435.

[9] P. Phan-Udom, N. Wattanakul, T. Sakulniwat, C. Ragkhitwetsagul, T. Sunetnanta, M. Choetkiertikul, and R. G. Kula, "Teddy: automatic recommendation of pythonic idiom usage for pull-based software projects," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 806–809.

[10] J. J. Merchante and G. Robles, "From python to pythonic: Searching for python idioms in github," in *Seminar Series on Advanced Techniques and Tools for Software Evolution (SATToSE)*, 2017.

[11] D. Orlov, "Finding Idioms in Source Code Using Subtree Counting Techniques," in *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*, 2020, pp. 44–54.

[12] P. Leelaprute, B. Chinthanet, S. Wattanakriengkrai, R. G. Kula, P. Jaisri, and T. Ishio, "Does coding in pythonic zen peak performance? preliminary experiments of nine pythonic idioms at scale," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 575–579.

[13] Z. Zhang, Z. Xing, X. Xia, X. Xu, L. Zhu, and Q. Lu, "Faster or slower? performance mystery of python idioms unveiled with empirical evidence," *arXiv preprint arXiv:2301.12633*, 2023.

[14] Z. Zhang, Z. Xing, X. Xia, X. Xu, and L. Zhu, "Making python code idiomatic by automatic refactoring non-idiomatic python code with pythonic idioms," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 696–708.

[15] J. Börstler, M. S. Hall, M. Nordström, J. H. Paterson, K. Sanders, C. Schulte, and L. Thomas, "An evaluation of object oriented example programs in introductory programming textbooks," *ACM SIGCSE Bulletin*, vol. 41, no. 4, pp. 126–143, 2010.

[16] M. de Raadt, R. Watson, and M. Toleman, "Textbooks: under inspection," University of Southern Queensland, Tech. Rep., 2005.

[17] W. J. Fitzgerald, J. Elmore, M. Kung, and A. J. Stenner, "The conceptual complexity of vocabulary in elementary-grades core science program textbooks," *Reading Research Quarterly*, vol. 52, no. 4, pp. 417–442, 2017.

[18] J. Börstler, M. Nordström, and J. H. Paterson, "On the quality of examples in introductory java textbooks," *ACM Trans. Comput. Educ.*, vol. 11, no. 1, feb 2011.

[19] S. F. Mazumder, C. Latulipe, and M. A. Pérez-Quiñones, "Are variable, array and object diagrams in java textbooks explanative?" in *Proceedings of the 2020 ACM conference on innovation and technology in computer science education*, 2020, pp. 425–431.

[20] M. Almansoori, J. Lam, E. Fang, A. G. Soosai Raj, and R. Chatterjee, "Textbook underflow: Insufficient security discussions in textbooks used for computer systems courses," in *Proceedings of the 52nd ACM technical symposium on computer science education*, 2021, pp. 1212–1218.

[21] I. Alpizar-Chacon, J. Barria-Pineda, K. Akhuseyinoglu, S. Sosnovsky, P. Brusilovsky *et al.*, "Integrating textbooks with smart interactive content for learning programming," in *CEUR Workshop Proceedings*, vol. 2895. CEUR WS, 2021, pp. 4–18.

[22] X. Huang, Q. Liu, C. Wang, H. Han, J. Ma, E. Chen, Y. Su, and S. Wang, "Constructing educational concept maps with multiple relationships from multi-source data," in *2019 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2019, pp. 1108–1113.

[23] J. Wu, Y. Sun, J. Zhang, Y. Zhou, and G. Huang, "A programming language learning service by linking stack overflow with textbooks," in *2023 IEEE International Conference on Web Services (ICWS)*. IEEE, 2023, pp. 234–245.

[24] G. Robles, R. G. Kula, C. Ragkhitwetsagul, T. Sakulniwat, K. Matsumoto, and J. M. Gonzalez-Barahona, "Pycefr: Python competency level through code analysis," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '22. New York, NY,

USA: Association for Computing Machinery, 2022, p. 173,Äì177. [Online]. Available: https://doi.org/10.1145/3524610.3527878

[25] T. Zimmermann, “Card-sorting: From text to themes,” in *Perspectives on Data Science for Software Engineering*, T. Menzies, L. Williams, and T. Zimmermann, Eds. Boston: Morgan Kaufmann, 2016, pp. 137–141. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128042069000271

[26] D. S. Cruzes and T. Dyba, “Recommended steps for thematic synthesis in software engineering,” in *2011 International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 275–284.

[27] F. Nelli, *Python Data Analytics: Data Analysis and Science using pandas, matplotlib and the Python Programming Language 1st Edition.* Apress; 1st edition, 2015.

[28] M. Summerfield, *Programming in Python 3 : a complete introduction to the Python language.* Upper Saddle River, New Jersey: Addison-Wesley, 2010.

[29] L. Ramalho, *Fluent Python: Clear, Concise, and Effective Programming.* O’Reilly Media, 1st edition, 2015.

[30] M. Venkitachalam, *Python Playground: Geeky Projects for the Curious Programmer 1st Edition.* No Starch Press, 1st edition, 2015.