Refactoring to Pythonic Idioms: A Hybrid Knowledge-Driven Approach Leveraging Large Language Models

ZEJUN ZHANG, Australian National University, Australia and CSIRO's Data61, Australia
ZHENCHANG XING, CSIRO's Data61, Australia and Australian National University, Australia
XIAOXUE REN[*], Zhejiang University, China
QINGHUA LU, CSIRO's Data61, Australia
XIWEI XU, CSIRO's Data61, Australia

Pythonic idioms are highly valued and widely used in the Python programming community. However, many Python users find it challenging to use Pythonic idioms. Adopting rule-based approach or LLM-only approach is not sufficient to overcome three persistent challenges of code idiomatization including code miss, wrong detection and wrong refactoring. Motivated by the determinism of rules and adaptability of LLMs, we propose a hybrid approach consisting of three modules. We not only write prompts to instruct LLMs to complete tasks, but we also invoke Analytic Rule Interfaces (ARIs) to accomplish tasks. The ARIs are Python code generated by prompting LLMs to generate code. We first construct a knowledge module with three elements including ASTscenario, ASTcomponent and Condition, and prompt LLMs to generate Python code for incorporation into an ARI library for subsequent use. After that, for any syntax-error-free Python code, we invoke ARIs from the ARI library to extract ASTcomponent from the ASTscenario, and then filter out ASTcomponent that does not meet the condition. Finally, we design prompts to instruct LLMs to abstract and idiomatize code, and then invoke ARIs from the ARI library to rewrite non-idiomatic code into the idiomatic code. Next, we conduct a comprehensive evaluation of our approach, RIdiom, and Prompt-LLM on nine established Pythonic idioms in RIdiom. Our approach exhibits superior accuracy, F1-score, and recall, while maintaining precision levels comparable to RIdiom, all of which consistently exceed or come close to 90% for each metric of each idiom. Lastly, we extend our evaluation to encompass four new Pythonic idioms. Our approach consistently outperforms Prompt-LLM, achieving metrics with values consistently exceeding 90% for accuracy, F1-score, precision, and recall.

CCS Concepts: • **Software and its engineering** → **Software evolution**.

Additional Key Words and Phrases: Pythonic Idioms, Large Language Model, Code Change

ACM Reference Format:
Zejun Zhang, Zhenchang Xing, Xiaoxue Ren, Qinghua Lu, and Xiwei Xu. 2024. Refactoring to Pythonic Idioms: A Hybrid Knowledge-Driven Approach Leveraging Large Language Models. *Proc. ACM Softw. Eng.* 1, FSE, Article 50 (July 2024), 22 pages. https://doi.org/10.1145/3643776

[*]Corresponding author.

Authors' addresses: Zejun Zhang, Australian National University, Canberra, Australia and CSIRO's Data61, Canberra, Australia, zejun.zhang@anu.edu.au; Zhenchang Xing, CSIRO's Data61, Canberra, Australia and Australian National University, Canberra, Australia, zhenchang.xing@data61.csiro.au; Xiaoxue Ren, Zhejiang University, Hangzhou, China, xxren@zju.edu.cn; Qinghua Lu, CSIRO's Data61, Sydney, Australia, qinghua.lu@data61.csiro.au; Xiwei Xu, CSIRO's Data61, Sydney, Australia, xiwei.xu@data61.csiro.au.

ACM 2994-970X/2024/7-ART50
https://doi.org/10.1145/3643776

1 INTRODUCTION

Pythonic idioms refer to programming practices and coding conventions that align with the core philosophy and style of the Python programming language [6, 16, 39, 50]. RIdiom [50, 52] identified nine Pythonic idioms by comparing syntax differences between Python and Java. Farooq et al. [16] conducted a literature review to identify and explore the usage of twenty-seven Pythonic idioms. An example of a Pythonic idiom is the chain-comparison idiom, which allows comparing multiple variables in one comparison operation, such as "-size <= x.indices(size)[0] <= size". The Python community continually strives to design and improve them to achieve code conciseness and improved performance [6, 13]. For the above example of chain-comparison, in contrast to the non-idiomatic equivalent, "-size <= x.indices(size)[0] and x.indices(size)[0] <= size", the chain-comparison simplifies code and improves performance. Given these benefits, the community and renowned Python developers actively promote the widespread adoption of Pythonic idioms [7, 8, 11, 20, 25, 39].

However, previous studies [6, 50] have indicated that Python users often be unaware of Pythonic idioms or unsure of how to correctly use Pythonic idioms, as Pythonic idioms are scattered across various materials, and are known for their versatile nature [7, 8, 11, 39]. For example, chain-comparison idiom also supports "in" operator (e.g., "line <= r[1] in rlist"), yet this usage often goes unnoticed by many Python users. To help Python users use Pythonic idioms, refactoring non-idiomatic code with Pythonic idioms emerges as a promising solution [35, 50]. Through pilot studies, we find the task faces three challenges including code miss, wrong detection and wrong refactoring because of the versatile nature of Pythonic idioms. Code miss refers to **missing non-idiomatic code that can be refactored with Pythonic idioms**. For example, RIdiom misses a For statement code that can be refactored with set comprehension, as shown in code ① of Figure 1. Wrong detection refers to **misidentifying non-refactorable non-idiomatic code with Pythonic idioms as refactorable**. For example, Prompt-LLM wrongly thinks a For statement without adding elements to a set as refactorable with set comprehension, as shown in code ⑤ of Figure 1. Wrong refactoring refers to **giving wrong idiomatic code for refactorable non-idiomatic code with Pythonic idioms**. For example, Prompt-LLM does not correctly refactor code "0< y_int <h_i and w_i < 0" with chain comparison, the corresponding idiomatic code is "w_i < 0 < y_int < h_i" as shown in code ③ of Figure 1.

A most related work, RIdiom [52], employs a rule-based approach to establish detection rules and refactoring procedures to refactor the non-idiomatic code into idiomatic code for nine Pythonic idioms. However, it is noteworthy that when confronted with intricate instances of non-idiomatic code, the reliance on pre-defined, inflexible rules cannot overcome the above three challenges (see RIdiom examples of Figure 1 in Section 2). Even when identified, formulating rules to refactor it into idiomatic code is challenging. On the other hand, in light of the success of Large Language Models (LLMs), users can simply describe natural language prompts to instruct LLMs to perform specific software engineering tasks, such as code generation [4, 15, 18, 32, 44] and program synthesis [21–23, 34]. It inspires us to explore LLMs for code idiomatization, wherein we observe their powerful ability in certain scenarios, e.g., code ① of Figure 1 can be correctly refactored with set comprehension by LLMs. However, without knowledge guiding, LLMs may make obvious mistakes that can be easily avoided using rules because of the inherent randomness and black boxes of LLMs (see Prompt-LLM examples of Figure 1 in Section 2).

This observation underscores the insufficiency of relying solely on rule-based approach or LLMs. It motivates us to propose a hybrid approach that combines the determinism inherent in rule-based approach with the adaptability offered by LLMs. Specifically, our hybrid approach comprises three core modules: **a knowledge module**, **an extraction module**, and **an idiomatization module**. For each module, we write prompts to instruct LLMs to complete tasks or invokes Analytic Rule

Interfaces (ARIs) to complete tasks. ARIs are Python code generated by prompting LLMs to generate code. The knowledge module is to construct a knowledge base consisting of three elements of non-idiomatic code of thirteen Pythonic idioms and an ARI library. The three elements are ASTscenario (the usage scenario of non-idiomatic code), ASTcomponent (the composition of non-idiomatic code) and Condition (the condition that refactorable non-idiomatic code must meet). The ARI library consists of ARIs to extract three elements and auxiliary ARIs to rewrite non-idiomatic code into idiomatic code. In the extraction module, for any syntax-error-free Python code, we invoke ARIs from the ARI library to extract ASTscenario and ASTcomponent that satisfies the condition, which will be input into the idiomatization module. The idiomatization module consists of three steps: abstracting code, idiomatizing code and rewriting code. We first abstractly represent the code of ASTcomponent by prompting LLMs. We then idiomatize the abstract code through LLM prompts, producing an abstract idiomatic code. Finally, we utilize ARIs to rewrite the non-idiomatic code into idiomatic code by using the abstract idiomatic code.

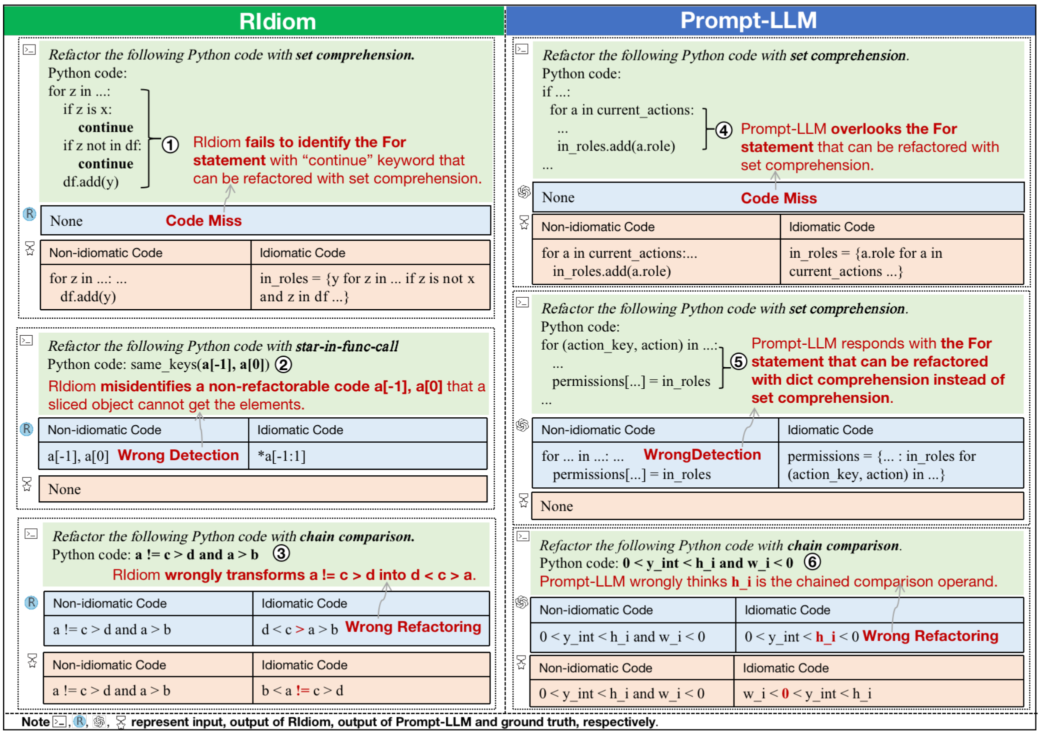
We conduct two experiments to evaluate the effectiveness and scalability of our approach. For effectiveness, we examine nine Pythonic idioms identified by RIdiom [52]. To determine a complete, correct and unbias benchmark, we randomly sample methods from the methods of each Pythonic idiom in RIdiom [52]. We independently run our approach, RIdiom and Prompt-LLM for the sampled methods to generate code pairs, and invite external workers to verify the correctness of code pairs by each approach manually. Then two authors and external workers discuss and resolve the inconsistencies. The metrics of accuracy, F1-score, precision, and recall were employed for evaluating results. The results demonstrate our approach achieves the best performance in accuracy, F1-score and recall compared to RIdiom and Prompt-LLM and achieves comparable precision with RIdiom. To evaluate the scalability of our approach, we choose four new Pythonic idioms not covered by RIdiom [52]. To avoid the bias of the benchmark, we randomly sample 600 methods from all methods in RIdiom [52]. To ensure the correctness and completeness of our approach, we following the same process in Section 4.1.2. Since RIdiom does not support idiomatization for the four Pythonic idioms. We do not run RIdiom on the new four idioms. Our approach consistently outperformed in accuracy, F1-score, precision, and recall, all surpassing 90% for each Pythonic idiom, which shows that our approach can be effectively extend to new Pythonic idioms.

In summary, the contributions of this paper are as follows:

- This is the first work to exploit LLMs into code idiomatization with Pythonic idioms, paving the way for new opportunities in code idiomatization.
- We propose a hybrid knowlege-driven approach with ARIs and prompts based on LLMs to refactor non-idiomatic code into idiomatic code with Pythonic idioms.
- We conduct experiments on both established and new Pythonic idioms. The high accuracy, F1-score, precision and recall verify the effectiveness and scalability of our approach. We provide a replication package [5] for future studies.

2 MOTIVATING EXAMPLES

Although using Pythonic idioms can improve the conciseness and performance [6, 26, 50, 51], refactoring non-idiomatic Python code with Pythonic idioms for a given Python code is not easy. RIdiom [52] is the state-of-the-art rule-based approach that formulates detection and refactoring rules to automatically refactor non-idiomatic code into idiomatic code for nine Pythonic idioms. Recently, large language models (LLMs) have achieved great success in various software engineering tasks [9, 17, 23, 33, 34, 36]. LLMs can directly complete various tasks by receiving natural language prompts as input, a process we refer to as Prompt-LLM. To explore challenges encountered in this endeavor for the two approaches, we randomly collect ten Python methods crawled by RIdiom [52]. The methods may contain several non-idiomatic code that can be refactored with a Pythonic idiom.



RIdiom

*Refactor the following Python code with **set comprehension.***
Python code:
```
for z in ...:
  if z is x:
    continue
  if z not in df:
    continue
  df.add(y)
```
① RIdiom **fails to identify the For statement** with "continue" keyword that can be refactored with set comprehension.

None	**Code Miss**

Non-idiomatic Code	Idiomatic Code
for z in ...: ... df.add(y)	in_roles = {y for z in ... if z is not x and z in df ...}

*Refactor the following Python code with **star-in-func-call***
Python code: same_keys(**a[-1], a[0]**) ②
RIdiom **misidentifies a non-refactorable code a[-1], a[0]** that a sliced object cannot get the elements.

Non-idiomatic Code	Idiomatic Code
a[-1], a[0] **Wrong Detection**	*a[-1:1]

None

*Refactor the following Python code with **chain comparison.***
Python code: **a != c > d and a > b** ③
RIdiom **wrongly transforms a != c > d into d < c > a.**

Non-idiomatic Code	Idiomatic Code
a != c > d and a > b	d < c > a > b **Wrong Refactoring**

Non-idiomatic Code	Idiomatic Code
a != c > d and a > b	b < a != c > d

Prompt-LLM

*Refactor the following Python code with **set comprehension**.*
Python code:
```
if ...:
  for a in current_actions:
    ...
    in_roles.add(a.role)
  ...
```
④ Prompt-LLM **overlooks the For statement** that can be refactored with set comprehension.

None	**Code Miss**

Non-idiomatic Code	Idiomatic Code
for a in current_actions:... in_roles.add(a.role)	in_roles = {a.role for a in current_actions ...}

*Refactor the following Python code with **set comprehension**.*
Python code:
```
for (action_key, action) in ...:
  ...
  permissions[...] = in_roles
...
```
⑤ Prompt-LLM responds with **the For statement that can be refactored with dict comprehension instead of set comprehension.**

Non-idiomatic Code	Idiomatic Code
for ... in ...: ... permissions[...] = in_roles **WrongDetection**	permissions = {... : in_roles for (action_key, action) in ...}

None

*Refactor the following Python code with **chain comparison**.*
Python code: **0 < y_int < h_i and w_i < 0** ⑥
Prompt-LLM wrongly thinks **h_i** is the chained comparison operand.

Non-idiomatic Code	Idiomatic Code
0 < y_int < h_i and w_i < 0	0 < y_int < **h_i** < 0 **Wrong Refactoring**

Non-idiomatic Code	Idiomatic Code
0 < y_int < h_i and w_i < 0	w_i < **0** < y_int < h_i

Note ⌨, Ⓡ, ֍, ⌛ **represent input, output of RIdiom, output of Prompt-LLM and ground truth, respectively.**

Fig. 1. Motivating examples

Next, we apply RIdiom and Prompt-LLM to each Python method for nine Pythonic idioms from RIdiom. Two authors collaborate to check results of RIdiom and Prompt-LLM and then identify challenges encountered by the two approaches. We summarize three challenges that are described as follows.

(1) Code Miss: miss non-idiomatic code that can be refactored with Pythonic idioms. The code written by Python users comes in various styles, the code may contain several refactorable non-idiomatic code with a Pythonic idiom and the form of non-idiomatic code may be diverse. Missing the refactorable non-idiomatic code can lead to redundant code and performance degradation. Unfortunately, code missing is common in the two approaches. On the one hand, given the diverse and intricate nature of non-idiomatic code patterns, some instances may pose challenges that surpass the capabilities of straightforward rule-based identification. For example, code ① of the RIdiom column of Figure 1 is a "for" statement with two "continue" statements that can be refactored with set-comprehension. Since set-comprehension does not support "continue" keyword, RIdiom wrongly assumes the code cannot be refactored. Actually, we can change "z is x" and "z not in df" into "z is not x" and "z in df", and then we use the "and" to connect the two conditions to remove continue statements. On the other hand, in a codebase, instances of non-idiomatic code are distributed throughout, necessitating a comprehensive scan of the entire codebase to identify such occurrences. Unlike rule-based programs that deterministically scan Python code from start to end, LLMs operate as black boxes. This non-deterministic nature can inadvertently lead to the oversight of refactorable non-idiomatic code [6, 16, 51]. For example, code ④ of the Prompt-LLM column in Figure 1 shows LLMs miss a "for" statement that can be refactored with set-comprehension.

(2) Wrong Detection: misidentify non-refactorable non-idiomatic code with Pythonic idioms as refactorable, which can lead to misunderstandings among Python users regarding

Pythonic idioms and potentially introducing bugs into the codebase. Although not common in RIdiom, it should not be ignored. The detection rules of RIdiom are human-defined, and developers may overlook the nuances of code structures and Python syntax semantics, leading to false discoveries. For example, for ② of Figure 1, RIdiom determines "-1, 0" is an arithmetic sequence, so it wrongly thinks that "a[-1], a[0]" can be obtained by a sliced object "a[-1:1]". However, since Python list grows linearly and is not cyclic, as such slicing does not wrap (from end back to start going forward) as we expect, the "a[-1:1]" actually is empty. On the other hand, although LLM has powerful abilities, its flexibility and adaptability usually cause inappropriate or off-topic response. For example, code ⑤ of the Prompt-LLM column in Figure 1 shows that the Prompt-LLM wrongly classifies a "for" statement that can be refactored with dict-comprehension as refactorable non-idiomatic code with set-comprehension. Actually, we can add a condition to check whether the for statement has an "add" function call to filter out the wrong detection. For another example, non-idiomatic code of chain comparison should have two comparison operations. However, Prompt-LLM often mistakenly suggests refactoring a single comparison operation with chain comparison, even though it cannot be refactored in this way. For instance, when encountering a single comparison operation like "start is not None", Prompt-LLM wrongly assumes it can be refactored using chain comparison.
(3) Wrong Refactoring: give wrong idiomatic code for refactorable non-idiomatic code with Pythonic idioms. It occurs in identifying refactorable non-idiomatic code but wrongly refactoring it, resulting in inconsistency in code behavior before and after refactoring. The diversity and complexity of such non-idiomatic code make both the rule-based approach and the Prompt-LLM approach prone to errors. For example, for the chain-comparison, to chain two comparison operations into one comparison operation, we need to reverse compare operands for each comparison operation and consider if we need to change the comparison operation. When one comparison operation has more than one comparison operation, it is more likely to make mistakes. For example, code ③ of the RIdiom column in Figure 1 shows that RIdiom wrongly transforms "a != c > d" into "d < c > a". Directly using LLMs may make unexpected mistakes. For example, code ⑥ of Prompt-LLM column of Figure 1 shows that Prompt-LLM assumes that "h_i" is the chained comparison operand and then wrongly refactors it into "0 < y_int < h_i < 0".

The three challenges shown in Figure 1 indicate that the rule-based approach, while deterministic, may still fall short in identifying all refactorable non-idiomatic code instances, especially those that are inherently complex or difficult to address through formulating rules (e.g., ① of Figure 1). Conversely, relying solely on the flexibility and adaptability of LLMs without knowledge guidance can lead LLMs to make obvious mistakes. For example, refactorable non-idiomatic code with set comprehension should contain an "add" function call. Regrettably, code ⑤ of Figure 1 lacks this function call. The absence of this contextual knowledge leads LLMs to misidentify it can be refactored with set comprehension. Therefore, a judicious approach emerges: initially employing code to handle deterministic and simple tasks, and then leveraging LLMs to tackle the more challenging refactoring endeavors where the rule-based approach may struggle. This hybrid approach stands poised to offer a comprehensive and effective solution.

Make Python code idiomatic encounters three challenges, including code miss, wrong detection and wrong refactoring. Adopting only one approach (rule-based approach or LLMs) alone is not sufficient.

3 APPROACH

Inspired by the motivating examples in Section 2, we propose a hybrid approach based on LLMs to refactor non-idiomatic code with Pythonic idioms. Figure 2 shows the approach overview. We first construct knowledge base of non-idiomatic code of Pythonic idioms, which consists of three elements: ASTscenario, ASTcomponent and Condition, and an ARI library consisting of ARIs to

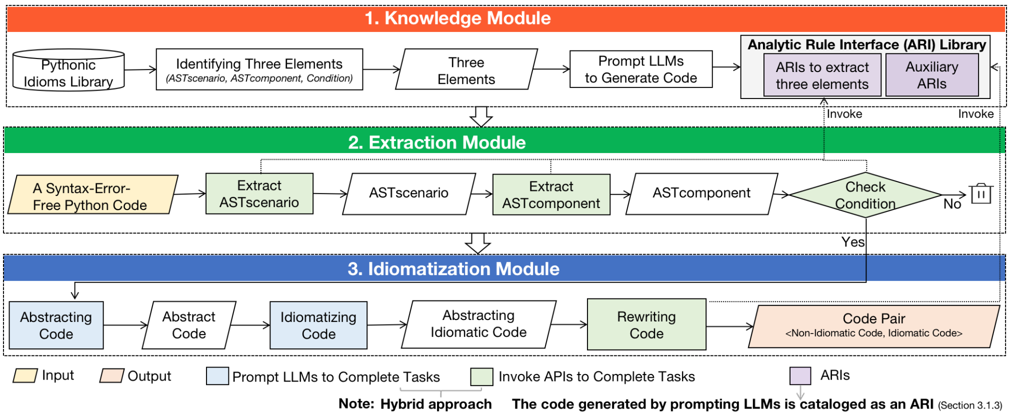

1. Knowledge Module
Pythonic Idioms Library
Identifying Three Elements (ASTscenario, ASTcomponent, Condition)
Three Elements
Prompt LLMs to Generate Code
Analytic Rule Interface (ARI) Library
ARIs to extract three elements
Auxiliary ARIs
Invoke
Invoke
2. Extraction Module
A Syntax-Error-Free Python Code
Extract ASTscenario
ASTscenario
Extract ASTcomponent
ASTcomponent
Check Condition
No
Yes
3. Idiomatization Module
Abstracting Code
Abstract Code
Idiomatizing Code
Abstracting Idiomatic Code
Rewriting Code
Code Pair <Non-Idiomatic Code, Idiomatic Code>
Input
Output
Prompt LLMs to Complete Tasks
Invoke APIs to Complete Tasks
ARIs
Note: Hybrid approach
The code generated by prompting LLMs is cataloged as an ARI (Section 3.1.3)

Fig. 2. Approach overview

extract the three elements and ARIs to rewrite code. After that, for a given Python code, we first call ARIs to extract its ASTcomponent from the ASTscenario, and then filter out ASTcomponent that does not meet the condition. Then we input the ASTcomponent and ASTscenario from extraction module into the idiomatization module. The idiomatization module consists of three steps: abstracting code, idiomatizing code and rewriting code. To reduce the pressure on LLMs to idiomatize code, we first abstract code by abstracting expression of the code corresponding ASTcomponent. And then we write prompts to make LLMs idiomatize the abstract code. After we get the abstract idiomatic code, we need to replace the abstract expression with the original expressions, and then rewrite the non-idiomatic code with the idiomatic code. Since the rewriting operations are simple, we call ARIs from the Auxiliary ARIs to complete.

3.1 Knowledge Module

As investigated in Section 2, lacking specific knowledge, directly using LLMs to find a non-idiomatic code with a Pythonic idiom from a given Python code is like finding a needle in a haystack. For example, the non-idiomatic code of set-comprehension should have a For node containing an add function call. Without the knowledge, Prompt-LLM misses one For node as shown in ④ of Figure 1, and misidentifies a For node without the add function call as non-idiomatic code of set-comprehension as shown in ⑤ of Figure 1. We observe the non-idiomatic code that can be refactored with Pythonic idioms has deterministic knowledge. Therefore, we can construct a knowledge base to boost the ability of LLMs.

3.1.1 Pythonic idioms library. Pythonic idioms are highly valued by developers [1, 20, 25], many studies summarize Pythonic idioms and research their usage [6, 16, 31, 50]. RIdiom [52] identified nine Pythonic idioms by comparing the syntax difference between Python and Java. The nine Pythonic idioms are list/set/dict-comprehension, chain-comparison, truth-test, loop-else, assign-multi-targets, for-multi-targets and star-in-func-call. State-of-the-art of research [16] based on a literature review identified a total of 27 detectable idioms, of which five (list/set/dict-comprehension, chain-comparison, truth-test) overlap with those defined by RIdiom. After excluding infrequently used idioms or those with rarely corresponding non-idiomatic Python code [1], four idioms remain: with, enumerate, fstring, and chain-assign-same-value. This culminates in a total of 13 Pythonic idioms. Table 1 gives the explanation and code examples of the 13 Pythonic idioms.

[1]For example, @staticmethod, assert and etc. are common syntax in programming languages, a few python developers use other syntax alone to achieve the same functionality without these idioms.

Table 1. Pythonic Idioms Library for Thirteen Pythonic Idioms

Source	Idiom	Explanation	Non-Idiomatic Code	Idiomatic Code
RIdiom [52], Farooq et al. [16]	list/set/dict -comprehension	Use one line to append elements to an iterable	**new_cols = []** **for col in old_cols:** **new_cols.append(col + postfix)**	**new_cols = [col + postfix for col in old_cols]**
	chain-comparison	Chain two comparison operations into one comparison operation [50]	**a > b and a < 1**	**b < a < 1**
	truth-test	Directly check the "truthiness" of an object	**embedding_dim % 2 == 0**	**not embedding_dim % 2**

RIdiom [52]	loop-else	A loop statement has an else clause	while attempt < 3:
			  ...
			  if body is not None:
			    break
			if body is None:
			  ...
				while attempt < 3:
				  ...
				  if body is not None:
				    break
				else:
				  ...

	assign-multi-tar	Assign multiple values to multiple variables in an assign statement	**self._ad = device** **self._sl4a_client = None**	**self._ad, self._sl4a_client = device, None**

	for-multi-tar	Unpack the iterated target of a for statement	for **sample** in family.samples:
			  if **sample[0]** > 2:
			  ...
				for **e0, *e** in family.samples:
				  if **e0** > 2:
				  ...

	star-in-func-call	Unpack an iterable to the positional arguments in a function call	nn.Linear(**gate_channels[i]** **, gate_channels[i+1])**	nn.Linear(***gate_channels[i: i + 2]**)
Farooq et al. [16]	with	Automatically close a file after it has been opened	bamfiles = [x.strip() for x in **open(bamfile)**]	**with open(bamfile) as f:** bamfiles = [x.strip() for x in f]

	enumerate	Return a tuple containing a count (from start which defaults to 0) and the values obtained from iterating over iterable.	for **i** in **range(len(text))**:
			  w = text[i]
			  if w in token2id:
			    R[i] = token2id[w]
				for **(i, w)** in **enumerate(text)**:
				  if w in token2id:
				    R[i] = token2id[w]

	chain-ass-same-value	Assign a value to multiple variables.	**global_draw_name = None** **_test_name = None**	**global_draw_name = _test_name =None**
	fstring	Dynamically combine data from variables and other data structures into a readable string output.	log.info('**sample_num_list is %s**' **% repr(self.sample_num_list))**	log.info(f'**sample_num_list is repr(self.sample_num_list)**')

3.1.2 *Three elements of non-idiomatic code of Pythonic idioms.* We construct the knowledge base of non-idiomatic code of Pythonic idioms as triples of <element, relation, element>. It comprises three fundamental elements: ASTscenario, ASTcomponent, and Condition. Each element focuses on a unique aspect: ASTscenario represents usage scenarios for non-idiomatic code linked to a Pythonic idiom. ASTcomponent defines the composition of such code, and Condition outlines necessary conditions for the ASTcomponents. There exist two relationships between these elements: the ASTcomponent relies on ASTscenario, and the ASTcomponent adheres to the specified conditions to be considered as non-idiomatic code of Pythonic idioms. Table 2 shows the three elements of non-idiomatic code of thirteen Pythonic idioms. The details are as follows:

ASTscenario: A non-idiomatic code associated with a Pythonic idiom may have restrictions on usage scenarios, corresponding to a distinct Abstract Syntax Tree (AST) node, referred to as ASTscenario. For example, chain-comparison idiom can chain two comparison operations using the "and" operator into one comparison. So the ASTscenario is a BoolOP node whose op is "and" as shown in Table 2. For another example, for the list comprehension idiom in Table 2, it allows the addition of elements to an object in just one line, as opposed to using a for statement. Since the for statement has no restrictions on the usage scenario, it does not possess an associated ASTscenario.

ASTcomponent: A non-idiomatic code associated with a Pythonic idiom has a deterministic composition, corresponding to few AST nodes, referred to as ASTcomponent. It serves as a pivotal entity in discerning and addressing non-idiomatic code patterns. Taking the chain-comparison idiom in Table 2 as an example, its ASTcomponent comprises two Compare nodes within a BoolOP node. These Compare nodes form essential elements of the non-idiomatic code pattern. For another example, for the list comprehension idiom in Table 2, which involves appending elements to an object in a for statement, its ASTcomponent is a For node and an Assign node.

Table 2. Three Elements of Non-Idiomatic Code of Thirteen Pythonic Idioms

Idiom	ASTscenario	ASTcomponent	Condition
list/set-comprehension	–	A For node An Assign node	1. The For node has "append/add" function call 2. The function name of "append/add" function call is the assigned variable of the Assign node
dict-comprehension	–	A For node An Assign node	1. The For node has an assign statement whose assigned variable is a Subscript node 2. The value of the Subscript node is the assigned variable of the Assign node
chain-comparison	A BoolOP node whose op is "and"	Two Compare nodes	1. Compare operands of the two Compare nodes intersect
truth-test	A test-type node	A Compare node	1. The op of the Compare node is "== "or "!=" 2. The one comparison operand is belong to EmptySet
loop-else	–	A For/While node An If node	1. The For/While node has break statements 2. The If node is the next statement of For node
assign-multi-tar	–	Consecutive Assign nodes	–
for-multi-tar	–	A For node	1. The body of the For node has a Subscript node 2. The value of the Subscript node is the iterated variable of the For node
star-in-func-call	A Call node	Consecutive Subscript nodes	1. The values of Subscript nodes are the same
with	–	A Call node	1. The function name of the Call node is "open"
enumerate	–	A For node	1. The iterated object is not a function call whose function name is "enumerate"
chain-ass-same-value	–	Consecutive Assign nodes	1. The values of consecutive Assign nodes are the same
fstring	–	A BinOP node	1. The op of the BinOp node is "%"

Condition: A non-idiomatic code associated with a Pythonic idiom may entail specific conditions, referred to as Condition for its ASTcomponent. It serves as a guiding principle for identification of refactorable ASTcomponent and avoid LLMs from idiomatizing non-refactorable ones that does not meet the specified conditions. For example, for the chain-comparison idiom in Table 2, the condition stipulates that the compare operands of the two Compare nodes must intersect. For another example, for the list-comprehension in Table 2, its non-idiomatic code is to append elements to a list, so the For node of ASTcomponent should has a "append" function call whose function name is the assigned variable of the Assign node.
Relation: There are two relationships between the three elements. The ASTcomponent is depend on the ASTscenario, the ASTcomponent should satisfy the Condition. For example, for the chain-comparison idiom, its AST component (two Compare nodes) is depend on the ASTscenario (BoolOp node whose op is "and"), and its ASTcomponent satisfies the Condition (Compare operands of the two Compare nodes intersect). These relationships establish a clear framework for identifying non-idiomatic code. We elaborate it in Section 3.2.

3.1.3 ARI library. The Analytic Rule Interface (ARI) library consists of ARIs to extract three elements in Table 2 and auxiliary ARIs. In contrast to directly relying on prompts to instruct LLMs in extracting the three essential elements from a given Python code, we employ LLMs to generate code to implement the required functionality. This approach addresses two key considerations. Firstly, within a project, it may have thousands of lines of code, and the non-idiomatic code is a small part of it. It is expensive for LLMs to handle so many codes and is difficult to make sure the non-idiomatic code is not missing and is correct from unrelated code in a given Python code. Secondly, by generating code through a single invocation of LLMs, we establish a reusable ARI library that can be leveraged consistently across different given Python code. Figure 3 shows examples to prompt LLMs to generate code.
Prompt LLMs to generate ARIs to extract three elements: We first create three prompt templates for the three elements: ASTscenario, ASTcomponent and Condition. Then we instantiate the prompt templates with three elements of each Pythonic idiom from Table 2. Finally, we instruct the LLM to generate code. Following the retrieval of the generated Python code, authors manually validate its correctness. Once verified, the code is cataloged as a reusable ARI, poised for application

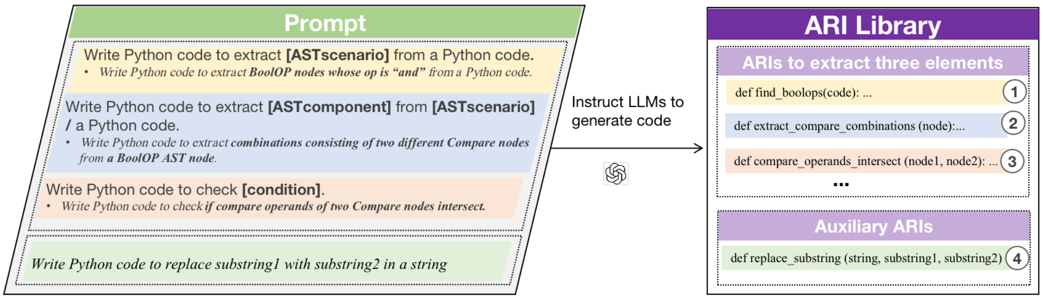

Prompt
Write Python code to extract [ASTscenario] from a Python code.
• Write Python code to extract BoolOP nodes whose op is "and" from a Python code.
Write Python code to extract [ASTcomponent] from [ASTscenario] / a Python code.
• Write Python code to extract combinations consisting of two different Compare nodes from a BoolOP AST node.
Write Python code to check [condition].
• Write Python code to check if compare operands of two Compare nodes intersect.
Write Python code to replace substring1 with substring2 in a string
Instruct LLMs to generate code
ARI Library
ARIs to extract three elements
def find_boolops(code): ...
def extract_compare_combinations (node):...
def compare_operands_intersect (node1, node2): ...
Auxiliary ARIs
def replace_substring (string, substring1, substring2)
1
2
3
4

Fig. 3. ARI library built by prompting LLMs to generate code

in subsequent any Python code [2]. This systematic approach ensures the reliability and reusability of the generated code for element extraction.

The template of ASTscenario is "Write Python method code to extract [ASTscenario] from a Python code", The [ASTscenario] is a placeholder which corresponds to the ASTscenario of a Pythonic idiom in Table 2. For example, for the chain-comparison idiom in Figure 3, the template is instantiated into "*Write Python method code to extract BoolOP nodes whose op is "and"*." When the prompt is input into the LLM, the LLM responds with an ARI called "extract_and_boolops(code)".

The template of ASTcomponent is "Write Python method code to extract [ASTcomponent] from [ASTscenario] / a Python code". The [ASTcomponent] and [ASTscenario] are placeholders which corresponds to the ASTscenario and ASTcomponent of a Pythonic idiom in Table 2. For example, for the chain-comparison in Figure 3 and Table 2, its ASTscenario is present, the template is instantiated into "*Write Python method code to extract combinations consisting of two different Compare nodes from a BoolOP node*". When the prompt is input into the LLM, the LLM responds with an ARI called "extract_compare_combinations(node)". For another example, for the list-comprehension, the ASTscenario is absent as shown in Table 2, the the template is instantiated into "*Write Python method code to extract For nodes from a Python code*". When the prompt is input into the LLM, the LLM responds with an ARI called "extract_for_nodes(code)".

The template of Condition is "Write Python method code to check [condition]". The [condition] is a placeholder which corresponds to the Condition of a Pythonic idiom in Table 2. For example, for the chain-comparison in Figure 3, the template is instantiated into "*Write Python method code to check if compare operands of two Compare nodes intersect*". When the prompt is input into the LLM, the LLM responds with an ARI called "compare_operands_intersect(node1, node2)". **Prompt LLMs to generate auxiliary ARIs:** For the auxiliary ARIs, it is a replace operation utilized in the idiomatization module. Since the replace operation is a simple task, we do not need to instruct LLMs to complete for each given Python code. And invocating the ARI can correctly and effectively replace substring1 with substring2 in a string. Specifically, we input the prompt "*Write Python method code to replace substring1 with substring2 in a string*" into the LLM, the LLM responds with "replace_substring(string, substring1, substring2)" whose body is "return string.replace(substring1, substring2)".

3.2 Extraction Module

After we construct the knowledge base of three elements of non-idiomatic code of Pythonic idioms and ARIs to extract three elements. We call ARIs from ARI library in order ASTscenario, ASTcomponent and Condition. To elaborate, if the ASTscenario exists, we first extract ASTscenario from a Python code, and then extract the ASTcomponent from the ASTscenario followed by filtering out components that do not satisfy the condition if condition exists. If the ASTscenario

[2]We manually verify that all ARIs are correct

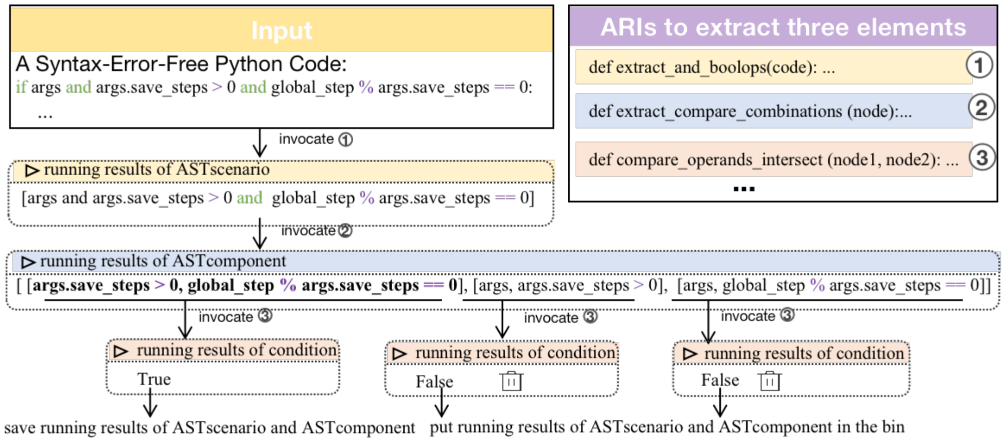

Input
A Syntax-Error-Free Python Code:
if args and args.save_steps > 0 and global_step % args.save_steps == 0:
...
ARIs to extract three elements
def extract_and_boolops(code): ... ①
def extract_compare_combinations (node):... ②
def compare_operands_intersect (node1, node2): ... ③
...
invocate ①
▷ running results of ASTscenario
[args and args.save_steps > 0 and global_step % args.save_steps == 0]
invocate ②
▷ running results of ASTcomponent
[[args.save_steps > 0, global_step % args.save_steps == 0], [args, args.save_steps > 0], [args, global_step % args.save_steps == 0]]
invocate ③
▷ running results of condition
True
False
False
save running results of ASTscenario and ASTcomponent
put running results of ASTscenario and ASTcomponent in the bin

Fig. 4. Examples of extraction module

does not exist, we directly extract the ASTcomponent from a Python code followed by filtering out ASTcomponent that does not satisfy the condition if the condition exists. For example, for the chain-comparison of Figure 4, we first call "extract_and_boolops" ARI to get all BoolOP nodes from a Python code. Then, for each BoolOP node, we call "extract_compare_combinations" ARI to extract all two different Compare nodes from the BoolOP node. Finally, for each two Compare nodes, we call "compare_operands_intersect" ARI to filter out two Compare nodes without common comparison operands. For another example, for the list-comprehension, its ASTscenario is empty, we directly call "extract_for_nodes(code)" to extract all For nodes from a Python code, and then we call "has_append(node)" filter out For nodes without "append" function call whose function name is the assigned variable of the Assign node.

3.3 Idiomatization Module

After extracting the code of ASTscenario and ASTcomponent, we can do the idiomatization task. Since the diversity and complexity of refactorable non-idiomatic code, it is not easy to complete the task by formulating rules. For example, consider code ① Figure 1. To refactor the code containing two continue statement with set-comprehension, we need to change "z is x" and "z not in df" into "z is not x" and "z in df", and then we use the "and" to connect the two conditions. Finally, we transform it with set-comprehension. Similarly, in the case of ⑥ of Figure 1, to refactor non-idiomatic code with chain-comparison, we need to reverse compare operands and determine whether need to change comparison operators based on the code semantic. Fortunately, LLMs trained on large corpora have rich knowledge and huge potential to complete complex tasks with natural language prompts [9, 17, 23, 33, 34]. Therefore, we write prompts to instruct LLMs to transform non-idiomatic code into idiomatic code for Pythonic idioms. To correctly complete the refactoring task, we design three steps including abstracting code, idiomatizing code and rewriting code.

3.3.1 Abstracting Code. The initial step involves the abstraction of the code which corresponds to ASTcomponent, wherein we keep related code snippets with Pythonic idioms while abstract unrelated code snippets with Pythonic idioms. By distilling the code to its core elements, we enhance the clarity and simplicity of the subsequent idiomatization process. To determine the abstract form for each Pythonic idiom, we invite three external workers with more than five years Python programming experience. We first introduce Pythonic idioms to make sure they are familiar with the idioms. Each worker is then asked to independently review the non-idiomatic code dataset of Pythonic idioms from previous researches [16, 50], and then writes a description of abstracting

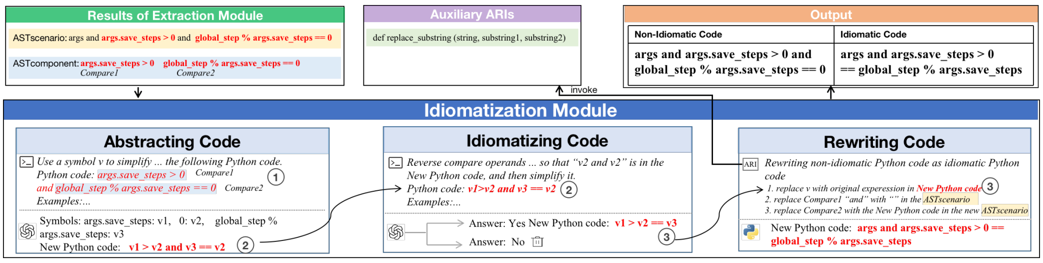

Results of Extraction Module
ASTscenario: args and args.save_steps > 0 and global_step % args.save_steps == 0
ASTcomponent: args.save_steps > 0 global_step % args.save_steps == 0
Compare1
Compare2
Auxiliary ARIs
def replace_substring (string, substring1, substring2)
Output
Non-Idiomatic Code
args and args.save_steps > 0 and global_step % args.save_steps == 0
Idiomatic Code
args and args.save_steps > 0 == global_step % args.save_steps
invoke
Idiomatization Module
Abstracting Code
Use a symbol v to simplify ... the following Python code.
Python code: args.save_steps > 0 Compare1
and global_step % args.save_steps == 0 Compare2
Examples:...
1
Symbols: args.save_steps: v1, 0: v2, global_step % args.save_steps: v3
New Python code: v1 > v2 and v3 == v2
2
Idiomatizing Code
Reverse compare operands ... so that "v2 and v2" is in the New Python code, and then simplify it.
Python code: v1>v2 and v3 == v2
2
Examples:...
Answer: Yes New Python code: v1 > v2 == v3
Answer: No
3
Rewriting Code
ARI
Rewriting non-idiomatic Python code as idiomatic Python code
1. replace v with original experession in New Python code
3
2. replace Compare1 "and" with "" in the ASTscenario
3. replace Compare2 with the New Python code in the new ASTscenario
New Python code: args and args.save_steps > 0 == global_step % args.save_steps

Fig. 5. Examples of idiomatization module

code and provides three examples of original non-idiomatic code and abstract non-idiomatic code. Then two authors discuss their results and give the final description of abstracting code (<*prompt*>) and three examples of original non-idiomatic code and the corresponding abstract non-idiomatic code for each Pythonic idiom (<*examples*>).

For the abstracting code for each Pythonic idiom, if the prompt has a specified object to abstract, we use ARIs from Auxiliary ARIs in Section 3.1.3 to replace a given string with another given string. Otherwise, we use prompt to instruct LLMs to complete the task. For example, for the star-in-func-call, the extracted non-idiomatic code is "feat.shape[-2], feat.shape[-1]" and the abstracted expression is a specified object ("feat.shape"). Therefore, we invoke "replace_substring" from auxiliary ARIs to replace "feat.shape"with "v", so the abstract code is "v[-2], v[-1]". For another example in the Abstracting Code of Figure 5, it does not have a specified object to abstract, We use prompt "*Use a symbol v to simplify each comparison operand within the following Python code. The same comparison operand is represented by the same symbol.*" to abstract represent the code "args.save_steps > 0 and global_step % args.save_steps == 0". The LLM responds with a symbol mapping and the abstract Python code "v1 > v2 and v3 == v2", facilitating subsequent idiomatizing process.

3.3.2 *Idiomatizing code.*

Building upon the abstracted Python code representation, this step focuses on the actual idiomatization of the code. To determine the prompt of idomatizing code for each Pythonic idiom, following the similar process 3.3.1, we invite the same three external works to further independently write a description to idiomatize the abstract code for each Pythonic idiom and provide examples with the abstract code and the corresponding idiomatic code for each Pythonic idiom. And then two authors discuss their results to give the final description of idiomatizing code (<*prompt*>), and the abstract code and the corresponding idiomatic code for each Pythonic idiom (<*examples*>).

For example, for the chain-comparison in the Idiomatizating Code of Figure 5, we use prompt "*Reverse compare operands of the first comparison operation, the second comparison, or the first and the second comparison operations so that "v2 and v2" is in the new Python code, and then simplify it*" to idiomatize the abstract Python code: "v1 > v2 and v3 == v2". The LLM responds with Yes and the abstract idiomatic Python code "v1 > v2 == v3". For another example, for the abstract code of chain-comparison "v1 in v2 and v3 in v2", The LLM responds with No because reversing compare operands is invalid for the "in" operator that can change the code semantic.

3.3.3 *Rewriting code.*

Following the acquisition of abstract idiomatic code from the idiomatization process, the final step involves rewriting the non-idiomatic code. It is achieved through the application of the "replace" ARI sourced from the Auxiliary ARIs. The "replace" operation serves a dual purpose: it facilitates the restoration of the abstract idiomatic code and allows for direct code rewriting by replacing the ASTcomponent with the idiomatic code in the ASTscenario, if

ASTscenario exists. Therefore, it may invoke "replace" several times. To determine the process of invoking "replace", following the similiar process in Section 3.3.1, we invite the same three external workers to further independently summarize steps to use "replace" to complete rewriting non-idiomatic code into idiomatic code. And then two authors discuss their results to give the final steps of rewriting code for each Pythonic idiom.

For example, for the Rewriting Code process illustrated in Figure 5, we first replace abstract symbols with their corresponding original expressions within the abstract idiomatic code: "v1 > v2 ==v3", yielding the genuine idiomatic code. Subsequently, we replace the Compare1 and 'and' with an empty string in the ASTscenario code, yielding the new ASTscenario code. Finally, we replace the Compare2 with the genuine idiomatic code in the new ASTscenario code, yielding a final idiomatic code: "args and args.save_steps > 0 == global_step % args.save_steps". The rewriting step ensures the precise transformation of non-idiomatic code into its idiomatic counterpart.

4 EVALUATION

To evaluate our approach, we study two research questions:

RQ1 (Effectiveness): What is the effectiveness of our approach in refactoring non-idiomatic Python code into idiomatic Python code with nine Pythonic idioms?
RQ2 (Scalability): Can our approach be effectively extended to new Pythonic idioms?

4.1 RQ1: Effectiveness of Refactoring Non-Idiomatic Python Code with Nine Pythonic Idioms

4.1.1 Motivation. RIdiom [50] can automatically refactor non-idiomatic code into idiomatic code with nine Pythonic idioms by formulating detection and refactoring rules, but it causes huge human investment in formulating rules. The current success of ChatGPT [3] demonstrates remarkable ability of LLMs to comprehend human prompts and complete the corresponding tasks. Therefore, we are interested in understanding the performance of our approach based on LLMs.

4.1.2 Approach. To clarify the effectiveness of our approach, we perform effectiveness comparison by calculating metrics on a dataset with our approach and the state-of-the-art baselines.
DataSet. To evaluate the effectiveness of our approach, it is fundamental to have a correct and complete benchmark of code refactorings consisting of code pairs <non-idiomatic Python code, idiomatic Python code>. Manually constructing a complete and correct benchmark is unrealistic because code refactoring involves a lot of time and manpower, and inevitably comes with personal bias. Recently, RIdiom [52] can automatically refactor non-idiomatic Python code with nine Pythonic idioms, which provides a good starting point. It provides code pairs within crawled methods for each Pythonic idiom. Considering the effort of manual verification, we randomly sample methods with a confidence level of 95% and a confidence interval of 5 from RIdiom [52] for each Pythonic idiom. Then, to ensure the completeness of the benchmark, we run RIdiom [52], our approach and Prompt-LLM on the sample methods for nine Pythonic idioms to collect code pairs <non-idiomatic Python code, idiomatic Python code>. To validate the correctness, we invite 18 external workers with more than five years Python programming experience. We divide them into 9 groups, and each group of two workers independently checks the correctness of the code pairs for an idiom. The Cohen's Kappa values [45] of nine groups for their annotation results all exceeded 0.75 (substantial agreement). Finally, the two authors and external workers discuss and resolve the inconsistencies and ensure the correctness of the benchmark for nine Pythonic idioms. Table 3 shows the benchmark of nine Pythonic idioms. The *Method* column represents the number of sampled methods for each Pythonic idiom, and the *Code Pair* column represents the number of code pairs <non-idiomatic code, idiomatic code> for each Pythonic idiom. Since sampled methods for each Pythonic idiom are

Table 3. Benchmark of Nine Pythonic Idioms

Idiom	Method	Code Pair
list-comprehension	389	512
set-comprehension	305	361
dict-comprehension	370	448
chain-comparison	391	574
truth-test	394	610
loop-else	319	360
assign-multi-targets	395	729
for-multi-targets	370	463
star-in-func-call	381	621
Total	3311	4678

from methods of RIdiom, it is reasonable to expect that the number of code pairs is greater than the number of the corresponding methods for each Pythonic idiom.

Baselines. We compare our approach with two baselines. The first baseline is RIdiom that is an approach based on rules proposed by Zhang et al. [50]. It detects and refactors non-idiomatic Python code into the corresponding idiomatic Python code with nine Pythonic idioms by manually formulating detection rules and refactoring steps. The second (Prompt-LLM) is to directly call the LLM to find code pairs for any given method code for each Pythonic idiom to illustrate the capability of LLM and the strengths of our proposed approach using LLM. For fairness of comparison, similar to process in Section 3.3.1, we invite three external works to independently write a prompt and provide three examples of a Python code and the corresponding code pairs. Then two authors discuss their results and give the final prompt and three examples for each Pythonic idiom. Examples are shown in Prompt-LLM column of Figure 1. Since the success of ChatGPT, the LLM our approach and baselines use are state-of-the-art of GPT-3.5-turbo [2]. And we set temperature to 0 to make the outputs mostly deterministic.

Metrics. By referring metrics using by previous researches on code refactorings [14, 38, 43, 50], we use four metrics accuracy, F1-score, precision and recall. To calculate the four metrics, we need to define true positives, false positives and false negatives which are represented as TP, FP and FN, respectively. We define a TP as a code pair detected by an approach is in the benchmark. We define a FP as a code pair detected by an approach is not in the benchmark. We define a FN as a code pair of the benchmark is not determined by an approach. The accuracy and F1-score represents the overall performance. We calculate accuracy, F1-score, precision and recall as follows:

$$Precision = \frac{TP}{TP + FP}, \quad Recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 * P * R}{P + R}, \quad Accuracy = \frac{TP}{TP + FP + FN}$$

4.1.3 Result. Figure 6 presents the accuracy, F1-score, precision and recall of our approach, RIdiom, Prompt-LLM for nine Pythonic idioms.

Comparison with RIdiom. In comparison to RIdiom, our approach consistently outperforms in three metrics: accuracy, F1-score, and recall across all Pythonic idioms. Notably, our approach exhibits a distinct advantage in both recall and accuracy over RIdiom. For eight Pythonic idioms, the accuracy and the recall exceed 90%. While our approach registers slightly below 90% in both recall and accuracy for the assign-multi-targets idiom, it is obviously close to 90% (87.8%). In contrast, RIdiom falls short of 90% in accuracy for five Pythonic idioms (list-comprehension, dict-comprehension, chain-comparison, loop-else, and assign-multi-targets), and four of these idioms (list-comprehension, dict-comprehension, chain-comparison, and assign-multi-targets) also have recall results below 90%. For the list-comprehension, the disparity in recall and accuracy

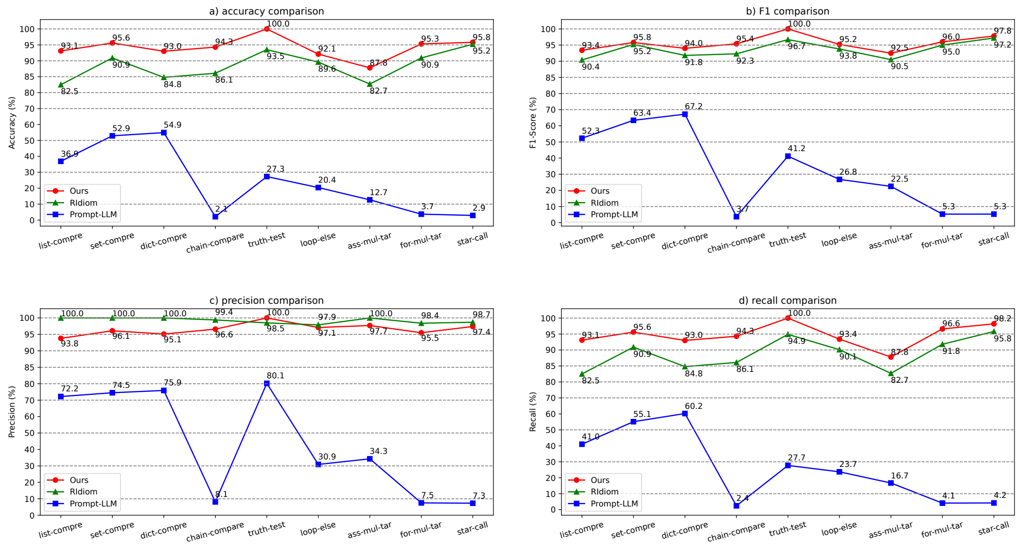


Fig. 6. Scatter plot with straight lines of accuracy, F1-score, precision and recall of three approaches for nine Pythonic idioms

between our approach and RIdiom exceeds 10%. For the other four idioms (dict-comprehension, chain-comparison, truth-test, and assign-multi-targets), these differences surpass 5%. For the remaining four idioms (set-comprehension, loop-else, for-multi-targets, and star-in-func-call), while the differences are relatively smaller, our approach maintains a consistent edge over RIdiom.

For the precision, although RIdiom exhibits a slight advantage in precision (with differences ranging from 0.8% to 6.2%) for eight of the idioms, our approach consistently achieves over 93.8% precision for each idiom. Moreover, our approach surpasses RIdiom for the truth-test idiom. It is important to note that F1-score strikes a balance between precision and recall, and in this regard, our approach consistently outperforms RIdiom across all Pythonic idioms. The comprehensive analysis underscores the notable advantages of our approach over RIdiom for the task of refactoring non-idiomatic code with Pythonic idioms, and can make up for the shortcomings of RIdiom in recall.

Comparison with Prompt-LLM. Compared to our approach, Prompt-LLM consistently exhibits the lowest performance across four metrics for each Pythonic idiom. The disparities between our approach and Prompt-LLM in terms of accuracy, F1-score, precision, and recall for the nine Pythonic idioms are substantial, ranging from 38.1% to 92.9%, 26.8% to 92.5%, 19.9% to 90.1%, and 32.8% to 94%, respectively.

Compared to Prompt-LLM, our approach maintains stable and commendable performance across all idioms, with each metric surpassing the 90% threshold for eight of them. Even in the case of assign-multi-targets, where both accuracy and recall fall slightly below 90%, they are still notably close at 90% (87.8%). In contrast, Prompt-LLM fails to achieve a metric score of 90% for any of the idioms.

Prompt-LLM demonstrates poor performance and exhibits significant variability across different Pythonic idioms. For list/set/dict-comprehension, Prompt-LLM displays relatively better performance across all metrics, exceeding 35%. This underscores LLMs' enhanced proficiency in handling the three idioms. For truth-test, Prompt-LLM exhibits superior precision (80.1%) compared to recall (27.7%), indicating a higher likelihood of missing refactorable non-idiomatic code instances associated with the truth-test idiom. For loop-else and assign-multi-targets, Prompt-LLM's performance remains limited across all metrics, falling below 35%. Furthermore, for the remaining three

Idiom	Non-idiomatic Code	Idiomatic Code	Failure Reason of Our Approach
list-comprehension	for i in range(len(unknowns) + 1): possible_mistakes.append(...) possible_mistakes.append(...)	possible_mistakes = [... for i in range(...)] + [... for i in range(...)]	Our approach gives the wrong idiomatic code of list-comprehension because it changes the order of appended elements
set-comprehenison	for u in urls_result: urls.add(u)	The given code is already simple and concise. Using set comprehension here would not make the code more readable or efficient.	Our approach abstains from refactoring with set-comprehension because LLM determines it would not enhance code conciseness

Fig. 7. The examples that our approach wrongly refactors or refrains from refactoring

idioms (chain-comparison, for-multi-targets, and star-in-func-call), Prompt-LLM's performance is markedly poorer, registering below 10% on all metrics. It suggests that Prompt-LLM struggles to effectively detect and refactor non-idiomatic code associated with the three idioms. Therefore, our approach significantly enhances the capabilities of LLMs, consistently demonstrating stable and commendable performance across all metrics.

Failure analysis of our approach. For the code pairs in the benchmarks that our approach does not find or wrongly refactor non-idiomatic Python code with nine Pythonic idioms, we summarize two reasons as follows:

(1) LLMs may produce suboptimal results when refactoring is too complicated. While LLMs have achieved success, it is reasonable that LLMs cannot handle all situations. For example, for the first example of Figure 7, the non-idiomatic code appends two different elements to the list "possible_mistakes" in each iteration of the for statement. Our approach finally gives the idiomatic code by concatenating two lists which independently appending two elements. The idiomatic code is wrong because the order of elements is different from the non-idiomatic code. Although the non-idiomatic code can be refactored with list-comprehension, the idiomatic code needs other statements to adjust the order of elements. For another example, for the non-idiomatic code "gnn_layer(..., n_points[idx1], n_points[idx2])", it is non-refactorable code with star-in-func-call because the "idx1, idx2" is not an arithmetic sequence. However, our approach mistakenly assumes the subscript sequence of "_points[idx1], n_points[idx2]" is an arithmetic sequence and refactor it into "gnn_layer(..., *n_points[idx1:idx2 + 1])" with star-in-func-call.

(2) LLMs may refrain from refactoring when benefits in idiomatic code appear limited: While LLM demonstrates proficiency in refactoring Python code using specific Pythonic idioms, there are instances where it abstains from doing so. For example, for the second example of Figure 7, the non-idiomatic code "for u in urls_results: urls.add(u)" actually can be refactored with set-comprehension. However, LLM responds with *"The given code is already simple and concise. Using set comprehension here would not make the code more readable or efficient. "*. LLM refuses to refactor it with set-comprehension because it may determine that applying set-comprehension would not notably enhance code conciseness. For another example, for the non-idiomatic code "slice2[axis] = slice(None, -1); slice1 = tuple(slice1)", LLM responds with *"The given code cannot be refactored with one assign statement as the variables being assigned are not of the same type"*. The LLMs refuses to refactor because it thinks "slice2[axis]" and "slice1" are not the same type.

Our approach can achieve best accuracy, F1-score and recall compared to other baselines. Although our approach performs slightly worse than RIdiom on precision, our approach can complement RIdiom on recall.

4.2 RQ2: Scalability of Our Approach

4.2.1 Motivation. Although RIdiom can achieve good result on nine Pythonic idiom, it cannot handle new Pythonic idioms and is difficult to extend to new Pythonic idioms because RIdiom needs manually formulate extracting and refactoring rules. So we are interested in whether our approach can be effectively extended to new Pythonic idioms that RIdiom cannot handle.

Table 4. Benchmark of Four New Pythonic Idioms

Idiom	Methods	Code Pairs
with	600	64
enumerate	600	565
chain-assign-same-value	600	156
fstring	600	223
Total	600	1008

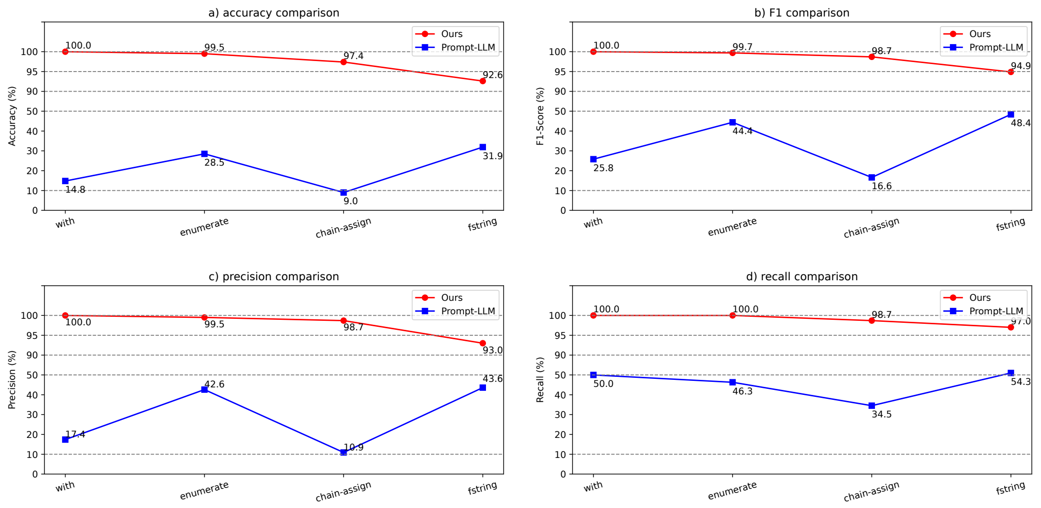


Fig. 8. Scatter plot with straight lines of accuracy, F1-score, precision and recall of two approaches for four new Pythonic idioms

4.2.2 Approach. Following the approach in Section 4.1.2, we present baselines, dataset and metrics. Baselines and metrics are the same as metrics and baselines in Section 4.1.2. Particularly, since RIdiom cannot handle the new four Pythonic idioms, the baseline is excluded. The dataset is detailed as follows: **DataSet.** As Section 3.1.1 illustrates, there are four new Pythonic idioms (with, enumerate, fstring and chain-assign-same-value) that RIdiom cannot handle. Considering that we need a certain number of code pairs (<non-idiomatic Python code, idiomatic Python code>) to evaluate our approach, but manually verifying code pairs is time consuming and the current GPT-3.5 is not free, we randomly sample 600 Python methods from crawled methods by RIdiom [50] that is a tool to refactor non-idiomatic code into idiomatic code with nine Pythonic idioms. Following the similar method in Section 4.1, to ensure the completeness of dataset as much as possible, we run our approach and Prompt-LLM on the sampled Python methods to collect code pairs <non-idiomatic Python code, idiomatic Python code>. To validate the correctness, we invite 8 external workers with more than five years Python programming experience. We divide them into four groups, and each group of two workers independently checks the correctness of the code pairs for an idiom. The Cohen's Kappa values [45] of four groups for their annotation results all exceeded 0.75 (substantical agreement). Finally, the two authors and external workers discuss and resolve the inconsistencies and ensure the correctness of the benchmark for the new four Pythonic idioms. Table 4 shows the benchmark of new four Pythonic idioms. The *Method* column represents the number of sampled methods for each Pythonic idiom, and the *Code Pair* column represents the number of code pairs <non-idiomatic code, idiomatic code> for each Pythonic idiom. The number of code pairs is less than the number of methods is reasonable because not each method may contain non-idiomatic code. For example, in the case of the with idiom, its non-idiomatic code should include file opening operations. However, not all methods necessarily involve file opening. Given that the with idiom is widely adopted by Python users [37], the count is relatively low (64). Conversely, for the enumerate idiom, its non-idiomatic code typically corresponds to a for statement, which is more prevalent in code. As a result, the count is relatively higher (565).

4.2.3 Result. Figure 8 presents the accuracy, F1-score, precision and recall of our approach, and Prompt-LLM for new four Pythonic idioms. For each Pythonic idioms, the metrics all are above 90%. And the accuracy, F1-score, precision and recall are above 95% for three idioms (with, enumerate and chain-assign-same-value). Compared to our approach, Prompt-LLM consistently exhibits poor performance across four metrics for each Pythonic idiom. The disparities between our approach and Prompt-LLM in terms of accuracy, F1-score, precision, and recall for the four Pythonic idioms are substantial, ranging from 60.7% to 88.4%, 46.5% to 82.1%, 49.4% to 87.8%, and 42.7% to 64.2%, respectively.

The high accuracy, F1-score, precison and recall of code refactorings of the four new Pythonic idioms illustrate that our approach can be effectively extended to other Pythonic idioms.

5 DISCUSSION

5.1 Implications

Our hybrid approach in Section 3 demonstrates excellent performance in Section 4, we now delve into the scalability of our approach and future work for researchers. Currently, our approach only supports syntactically correct Python code, as it requires parsing the code into an Abstract Syntax Tree (AST). However, handling Python code with syntax errors on the hybrid framework is feasible. One solution is to incorporate a syntax-fixing module to rectify syntax errors in the Python code before it input into extraction module in Section 3.2. Another approach is to introduce an alternative method within the extraction module in Section 3.2. Specifically, prompting LLMs to extract three elements or prompting LLMs to generate AST for Python code instead of invoking APIs, which can be effective when the syntax error cannot be fixed.

Prior studies [6, 16, 50, 51] have shown that employing Pythonic idioms may yield benefits, such as code conciseness and improved performance, but it can also have drawbacks, including potential impacts on code readability and performance degradation. Our current focus is solely on refactoring non-idiomatic code with Pythonic idioms. The results in Section 4 and Figure 7 highlight that LLMs may abstain from refactoring when benefits from idiomatic code are limited. This observation prompts researchers to consider using LLMs combined with knowledge base to generate comments explaining the positive and negative effects of refactoring non-idiomatic code with Pythonic idioms. This can enhance Python users' understanding and effective utilization of Pythonic idioms.

Furthermore, our approach combines ARIs and prompts based on LLMs to refactor non-idiomatic code with Pythonic idioms. While current Language Models (e.g., ChatGPT [33]) are not freely available, there has been a recent emergence of free alternatives (e.g., Llama 2 [42]). Over time, it is plausible that more LLMs may become more accessible, potentially even free of charge. This accessibility could assist developers in effectively refactoring code using Pythonic idioms to alleviate the limitation of rule-based approach.

5.2 Threats to Validity

Internal Validity: The one internal threat is inaccuracy when evaluating the correctness of benchmark in Section 4. For each Pythonic idiom, we invite two external workers with more than five Python programming experience. And two authors and external workes discuss to resolve the inconsistencies to ensure the correctness of the benchmark.

External Validity: One external threat is that our approach is limited to thirteen Pythonic idioms. RIdiom [52] can only handle nine Pythonic idioms, our approach contains four new Pythonic idioms which validate the scalability of our approach. When there are more Pythonic idioms, developers can determine the three elements of Pythonic idioms to extend to more Pythonic idioms. In the future, we will automatically mine undetected Pythonic idioms and automatically determine three elements of Pythonic idioms. The other external threat is the representative of the benchmark selected to evaluate our approach. To mitigate this threat, our experimented methods of benchmark are from previous research [52], representing an unbiased benchmark for our research.

6 RELATED WORK

Studies on Pythonic idioms. Pythonic idioms are highly valued by researchers, there are several studies [6, 16, 20, 25, 30, 40, 41, 52, 53] to mine Pythonic idioms or help Python users use Pythonic idioms better. Alexandru et al. [6] and Farooq et al. [16] conducted an independent literature review to create a catalogue of Pythonic idioms. There are 27 detectable Pythonic idioms involving built-in methods, APIs and syntax. Phan-udom et al. [35] first collected 58 non-idiomatic code instances and 55 idiomatic. Subsequently, they provided Pythonic code examples akin to code in projects of developer. Sakulniwat et al. [37] proposed a technique to visualize and understand the usage of the with Pythonic idiom and found developers tend to adopt the idiomatic code over time. Dilhara et al. [14] mined frequent code changes in Python ML systems and found some of involving Pythonic idioms. RIdiom [52] first identified nine Pythonic idioms by comparing the syntax difference between Python and Java and then designed detection and refactoring rules to automatically refactor non-idiomatic code into idiomatic code with the nine Pythonic idioms. Leelaprute et al. [26] analyzed the performance of Python features (e.g., collections.defaultdict and lambda) and two Pythonic idioms (list/dict comprehension) with different input sizes. Zhang et al. [51] conducted a large-scale empirical study for nine Pythonic idioms by creating a synthetic dataset and real-project dataset in RIdiom [52]. They found Pythonic idioms do not always result in performance speedup and can cause degraded performance. Zhang et al. [53] explored the challenges in comprehending Pythonic idioms, their conciseness, and the potential impact of comprehension challenges on code. They developed the DeIdiom tool to interpret these idioms into equivalent non-idiomatic code, facilitating developers in comprehending and effectively leveraging Pythonic idioms. In this work, we focus one refactoring non-idiomatic code with Pythonic idioms. Unlike RIdiom [52], we do not rely on a rule-based approach; instead, we employ a hybrid approach utilizing APIs and prompts based on LLMs. Besides, we extend our approach to four new Pythonic idioms, which further verify the scalability of our approach.

Studies on LLMs. Building on the achievements of Language Models (LLMs) like GPT-3 [9] and GPT-4 [33] in the field of Natural Language Processing [10, 12, 27, 36, 46–49], researchers are now delving into their potential applications in software engineering [4, 17, 19, 21, 23, 24, 28, 29, 34]. Huang et al. [23] introduced a chain-of-thought approach based on LLMs, comprising four steps: extracting structure hierarchy, isolating nested code blocks, generating CFGs for these nested blocks, and amalgamating all CFGs. This approach surpasses existing CFG tools in terms of both node and edge coverage, particularly for incomplete or erroneous code. Feng et al. [17] proposed a two-phase approach utilizing a chain-of-thought prompt to guide LLMs in extracting S2R entities, followed by matching these entities with GUI states to replicate the bug reproduction steps. This demonstrates that instructing LLMs through prompts can effectively achieve bug replay. Peng et al. [34] presented TYPEGEN, which generates prompts incorporating domain knowledge, then feeds them into LLMs for type prediction. This approach uses few annotated examples to achieve superior performance compared to rule-based type inference approaches. In this work, we focus on making Python code idiomatic with Pythonic idioms. Unlike previous approaches that exclusively design prompts for instructing LLMs to generate code or complete tasks, we propose a hybrid approach using APIs generated by LLMs to extract three elements, along with prompts to guide LLMs in performing the idiomatization task. It demonstrates that LLMs and rule-based approaches complement each other which can assist researchers to solve software engineering tasks better.

7 CONCLUSION AND FUTURE WORK

Refactoring non-idiomatic code with Pythonic idioms is not easy because of three challenges including code miss, wrong detection and wrong refactoring. Depending solely on Large Language Models (LLMs) or a rule-based approach (RIdiom) has its limitations in addressing these challenges. To alleviate the challenges, we propose a hybrid approach based on LLMs to refactor non-idiomatic code with Pythonic idioms. In detail, we first extract three elements of a Pythonic idiom and create an API library by prompting LLMs to generate code. We then invoke the APIs to extract non-idiomatic code in the extraction module. Finally, we prompt LLMs to abstract code, idiomatize the abstract code and then invoke APIs to rewrite non-idiomatic code into idiomatic code. The results of our experiments, conducted on nine Pythonic idioms from RIdiom [52], as well as four new Pythonic idioms [16] not covered by RIdiom, demonstrate high levels of accuracy, F1-score, precision, and recall. This substantiates the effectiveness and scalability of our proposed approach. In the future, we will keep improving our approach and extend our approach to accommodate Python code with syntax errors. Besides, we plan to offer explanations regarding the impacts, such as enhanced readability and performance, that result from refactoring code into idiomatic Python.

8 DATA AVAILABILITY

Our replication package can be found here [5].

REFERENCES

[1] 2022. *Programming Idioms.* https://programming-idioms.org/

[2] 2023. *GPT.* https://platform.openai.com/docs/guides/gpt

[3] 2023. *Introducing ChatGPT.* https://chat.openai.com/

[4] 2023. *OpenAI Codex.* https://openai.com/blog/openai-codex

[5] 2023. *Replication Package.* https://github.com/idiomaticrefactoring/IdiomatizationLLM

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

[7] D. Bader. 2017. *Python Tricks: A Buffet of Awesome Python Features.* BookBaby. https://books.google.co.in/books?id=C0VKDwAAQBAJ

[8] D. Beazley and B.K. Jones. 2013. *Python Cookbook: 3rd Edition.* O'Reilly Media, Incorporated. https://books.google.com.au/books?id=oBKwkgEACAAJ

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[10] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code Generation with Generated Tests. arXiv:2207.10397 [cs.CL]

[11] Quantified Code. 2014. *The Little Book of Python Anti-Patterns.* https://github.com/quantifiedcode/python-anti-patterns

[12] Antonia Creswell, Murray Shanahan, and Irina Higgins. 2022. Selection-Inference: Exploiting Large Language Models for Interpretable Logical Reasoning. https://doi.org/10.48550/arXiv.2205.09712

[13] Python developers. 2000. *Python Enhancement Proposals.* https://peps.python.org/pep-0000/

[14] Malinda Dilhara, Danny Dig, and Ameya Ketkar. 2023. PYEVOLVE: Automating Frequent Code Changes in Python ML Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* IEEE, 995–1007.

[15] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. arXiv:2304.07590 [cs.SE]

[16] Aamir Farooq and Vadim Zaytsev. 2021. There is More than One Way to Zen Your Python. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. 68–82.

[17] Sidong Feng and Chunyang Chen. 2023. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. arXiv:2306.01987 [cs.SE]

[18] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999 [cs.SE]

[19] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-Based Automated Software Vulnerability Repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 935–947. https://doi.org/10.1145/3540250.3549098

[20] Raymond Hettinger. 2013. *Transforming code into beautiful, idiomatic Python.* https://www.youtube.com/watch?v=OSGv2VnC0go

[21] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. Prompt-Tuned Code Language Model as a Neural Knowledge Base for Type Inference in Statically-Typed Partial Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 79, 13 pages. https://doi.org/10.1145/3551349.3556912

[22] Qing Huang, Jiahui Zhu, Zhenchang Xing, Huan Jin, Changjing Wang, and Xiwei Xu. 2023. A Chain of AI-based Solutions for Resolving FQNs and Fixing Syntax Errors in Partial Code. arXiv:2306.11981 [cs.SE]

[23] Qing Huang, Zhou Zou, Zhenchang Xing, Zhenkang Zuo, Xiwei Xu, and Qinghua Lu. 2023. AI Chain on Large Language Model for Unsupervised Control Flow Graph Generation for Statically-Typed Partial Code. arXiv:2306.00757 [cs.SE]

[24] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models Meet Program Synthesis. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3510003.3510203

[25] Jeff Knupp. 2013. *Writing Idiomatic Python 3.3.* Jeff Knupp.

[26] Pattara Leelaprute, Bodin Chinthanet, Supatsara Wattanakriengkrai, Raula Gaikovina Kula, Pongchai Jaisri, and Takashi Ishio. 2022. 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*. 575–579.

[27] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[28] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Ré mi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (dec 2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[29] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the Blank: Context-Aware Automated Text Input Generation for Mobile GUI Testing. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1355–1367. https://doi.org/10.1109/ICSE48619.2023.00119

[30] José J. Merchante. 2017. From Python to Pythonic: Searching for Python idioms in GitHub. https://api.semanticscholar.org/CorpusID:211530803

[31] José Javier Merchante and Gregorio Robles. 2017. From Python to Pythonic: Searching for Python idioms in GitHub. In *Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution*. 1–3.

[32] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Haiquan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *International Conference on Learning Representations*. https://api.semanticscholar.org/CorpusID:252668917

[33] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[34] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Generative Type Inference for Python. arXiv:2307.09163 [cs.SE]

[35] Purit Phan-udom, Naruedon Wattanakul, Tattiya Sakulniwat, Chaiyong Ragkhitwetsagul, Thanwadee Sunetnanta, Morakot Choetkiertikul, and Raula Gaikovina Kula. 2020. Teddy: Automatic Recommendation of Pythonic Idiom Usage For Pull-Based Software Projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 806–809.

[36] Xiaoxue Ren, Xinyuan Ye, Dehai Zhao, Zhenchang Xing, and Xiaohu Yang. 2023. From Misuse to Mastery: Enhancing Code Generation with Knowledge-Driven AI Chaining. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 976–987. https://doi.org/10.1109/ASE56229.2023.00143

[37] Tattiya Sakulniwat, Raula Gaikovina Kula, Chaiyong Ragkhitwetsagul, Morakot Choetkiertikul, Thanwadee Sunetnanta, Dong Wang, Takashi Ishio, and Kenichi Matsumoto. 2019. 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, 43–435.

[38] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 269–279.

[39] Brett Slatkin. 2020. *Effective Python : 90 specific ways to write better Python / Brett Slatkin.* (second edition. ed.). Addison-Wesley, Place of publication not identified.

[40] Mark Summerfield. 2009. *Programming in Python 3: A Complete Introduction to the Python Language* (2nd ed.). Addison-Wesley Professional.

[41] Balázs Szalontai, Ákos Kukucska, András Vadász, Balázs Pintér, and Tibor Gregorics. 2023. Localizing and Idiomatizing Nonidiomatic Python Code with Deep Learning. In *Intelligent Computing*, Kohei Arai (Ed.). Springer Nature Switzerland, Cham, 683–702.

[42] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[43] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering*. 483–494.

[44] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022, Extended Abstracts*, Simone D. J. Barbosa, Cliff Lampe, Caroline Appert, and David A. Shamma (Eds.). ACM, 332:1–332:7. https://doi.org/10.1145/3491101.3519665

[45] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding interobserver agreement: the kappa statistic. *Fam med* 37, 5 (2005), 360–363.

[46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]

[47] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 385, 22 pages. https://doi.org/10.1145/3491102.3517582

[48] Tongshuang Wu, Michael Terry, and Carrie J. Cai. 2022. AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts. arXiv:2110.01691 [cs.HC]

[49] Kevin Yang, Nanyun Peng, Yuandong Tian, and Dan Klein. 2022. Re3: Generating Longer Stories With Recursive Reprompting and Revision. In *Conference on Empirical Methods in Natural Language Processing*. https://api.semanticscholar.org/CorpusID:252873593

[50] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, and Liming Zhu. 2022. 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 (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 696–708. https://doi.org/10.1145/3540250.3549143

[51] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1495–1507. https://doi.org/10.1109/ICSE48619.2023.00130

[52] Zejun Zhang, Zhenchang Xing, Xiwei Xu, and Liming Zhu. 2023. RIdiom: Automatically Refactoring Non-Idiomatic Python Code with Pythonic Idioms. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion*

Proceedings (ICSE-Companion). IEEE, 102–106.

[53] Zejun Zhang, Zhenchang Xing, Dehai Zhao, Qinghua Lu, Xiwei Xu, and Liming Zhu. 2024. Hard to Read and Understand Pythonic Idioms? DeIdiom and Explain Them in Non-Idiomatic Equivalent Code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)* (, Lisbon, Portugal,) *(ICSE 2024)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3597503.3639101

Received 2023-09-29; accepted 2024-01-23