Proceedings Paper:
Evans, Abi orcid.org/0000-0001-8647-3690, Wang, Zihan, Liu, Jieren et al. (1 more author) (2023) SIDE-lib:A Library for Detecting Symptoms of Python Programming Misconceptions. In: Proceedings of the 28th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '23). 28th annual ACM conference on Innovation and Technology in Computer Science Education, 10-12 Jul 2023 ACM, FIN, 159–165.

https://doi.org/10.1145/3587102.3588838

SIDE-lib: A Library for Detecting Symptoms of Python Programming Misconceptions

Abigail C. Evans
abi.evans@york.ac.uk
University of York
York, UK

Zihan Wang
zihan2wang@gmail.com
Northeastern University
Seattle, USA

Jieren Liu*
liujieren95@gmail.com
Northeastern University
Seattle, USA

Mingming Zheng*
zheng.mingm@northeastern.edu
Northeastern University
Seattle, USA

ABSTRACT

Extensive prior work has identified and described misconceptions held by novice programmers. Much of this prior work has involved at least some automatic detection of potential misconceptions using a variety of methods such as intercepting compiler error messages, pattern matching, and black-box testing. To the best of our knowledge, no independent and flexible tool for automatic detection of misconceptions is currently available to the research community, meaning that detection must be reimplemented from scratch for each new project that aims to understand or support novice programmers using automatic analysis. This is time-consuming work, particularly for misconceptions that require understanding of the context of a program beyond localised syntax patterns. In this paper, we introduce SIDE-lib, a standalone library for detecting symptoms of Python misconceptions. This library is made available with the goal of simplifying and speeding up research on Python misconceptions and the development of tools to support learning. We also describe example use cases for the library, including how we are using it in our ongoing research.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**; • **Applied computing → Education**.

KEYWORDS

computer science education; novice programmers; library; misconceptions

ACM Reference Format:
Abigail C. Evans, Zihan Wang, Jieren Liu, and Mingming Zheng. 2023. SIDE-lib: A Library for Detecting Symptoms of Python Programming Misconceptions. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023), July 8–12, 2023, Turku, Finland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587102.3588838

*These authors contributed equally to this research.

ITiCSE 2023, July 8–12, 2023, Turku, Finland.

ACM ISBN 979-8-4007-0138-2/23/07…$15.00
https://doi.org/10.1145/3587102.3588838

1 INTRODUCTION

Beginner programmers' misconceptions have been the subject of decades of research [1–3, 8, 10–12, 15–17, 20–22, 28, 29, 33]. A subset of the research in this space involves automatic analysis of code such as testing programs for correct output [1, 11], comparing student work with pre-defined solutions or concept inventories [27, 33], and/or analysis of syntax and error messages [2, 5, 9, 13]. Despite the large volume of research on programming misconceptions and a small number of approaches to automatic analysis of code, there does not exist a standard tool to support misconception detection. Instead, researchers must implement their own detection methods leading to significant duplication of effort across projects.

As well as slowing down research on misconceptions themselves, the lack of shared resources for detecting misconceptions also hinders the development of new tools to support novices as detection of problems must be implemented before design work can begin. Our own long term goal is to embed support for novice Python programmers into mainstream IDEs. As part of this work, we developed a standalone, task-independent library for detecting symptoms of misconceptions in Python code, which we are making available to other researchers with the goal of simplifying some aspects of misconception research.

Our detection library is designed to be easy to install, simple to use, and free of external dependencies. It is also flexible enough to support a variety of use cases out of the box (e.g. research on misconceptions vs. student-facing learning interventions) in unconstrained programming environments.

In the following sections, we define what is meant by "symptom" and "misconception" in this work; we summarise past approaches to automatic detection of programming misconceptions; we describe the design and development of our library, which was the result of an analysis of 1331 Python programs written by novices; and we present example applications that make use of the library.

2 SYMPTOMS AND MISCONCEPTIONS

Substantial prior work has sought to identify and categorise programming errors and misconceptions held by novices. We use "error" to refer to any code that produces output that is not what the programmer intended (or what the task required), whether or not it

causes the program to crash [25]; and "misconception" to describe inaccuracies in a programmer's knowledge [29].

From Pea's [28] description of high level language-independent "bugs" in novices' mindset and approach, to the identification of language-specific errors [3, 5], error categorisation has received a lot of attention. Errors can be categorised as *syntactic*, *conceptual*, or *strategic* [1, 4, 26, 29]. Like [8], our work focuses on the first two error categories only. We exclude strategic errors as they are generally task dependent.

Syntactic errors are mistakes relating to the syntax rules of a language. Conceptual errors are misconceptions about a programming construct, such as how a loop works. Albrecht and Grabowski [1] add *sloppiness* errors, which are unintentional mistakes such as typos.

Broad categories are useful to understand the types of knowledge that learners struggle with but more specific code-level categories are needed to inform tools or teaching strategies to tackle misconceptions. Studies that enumerate low level errors and deeper misconceptions identify a varying number of categories e.g. 31 [7], 62 [18], 80 [1], 90 [24, 25], 162 [34]. This indicates that despite a considerable amount of work in this area, there is still no definitive list.

Kaczmarczyk et al. [22] argue that there is work to be done to understand novice's misconceptions themselves, not just the errors they cause. Recently, research efforts have shed more light on misconceptions in Python [15, 20], explored if and how documented misconceptions apply across programming languages (C, Python, and Java) [7], and led to the creation of extensive documentation on misconceptions organised by language [8].

Chiodini et al. [8] define a new subcategory of misconceptions, *programming language misconceptions*: statements that "can be disproved by reasoning entirely based on the syntax and/or semantics of a programming language". Programming language misconceptions are the targets of our detection library.

Although programming language misconceptions relate to language specifications, they are properties of learners, not programs or languages. In order to fully understand a learner's beliefs or mental model of a programming construct, we must observe what they do and/or say about that construct. However, as the goal of our library is to automatically detect misconceptions in an unconstrained environment without knowledge of the student's task, the library is restricted to observing code. Therefore, we focus on identifying potential misconceptions by looking for their *symptoms* rather than the misconceptions themselves. Symptoms are the manifestation of a learner's misconception in the code that they write; patterns in statements or blocks of code that suggest a possible misconception. Symptoms relate to both errors and misconceptions but these concepts are not interchangeable. Symptoms will often contain or produce errors but symptoms of some misconceptions will not lead to incorrect output.

In some cases, a single symptom is closely mapped to a single misconception. For example, a documented misconception is that a Boolean expression must be compared with a Boolean literal in order to check if it is true or false [8]. This misconception will manifest in a single symptom: the use of `"== True"` or `"== False"` in a Boolean expression. Given that this misconception results in redundant code but will not affect program output, the presence of this symptom in a learner's code does not conclusively prove that the learner holds the associated misconception—it may be a matter of stylistic preference or a desire for clarity.

The previous example shows how a single symptom in a learner's code may indicate a particular misconception. However, some misconceptions require the presence of multiple symptoms and a single symptom may be associated with more than one misconception. For example, we consider an unused return value a symptom, but it may indicate different misconceptions depending on the presence of other symptoms. Figure 1 shows two examples of an unused return, each indicating a different possible misconception.

```
1   def is_foo(word):
2     if word == word[::-1]:
3       print("Foo")
4       return True
5     else:
6       print("Bar")
7       return False
8
9   test = input("Enter a word")
10  test.lower()
11  is_foo(test)
```

Figure 1: A program that contains two examples of the symptom *UnusedReturnValue*: the expression on line 10, which returns a String, and the function call on line 11, which returns a Boolean. Neither value is saved or used. Each case indicates a different possible misconception.

The first unused return in Figure 1 is found on line 10, where a method is called to convert a String to lowercase but the converted String is not saved or used. In this example, the unused return may indicate a belief that String methods modify the original String. The second unused return occurs on line 11, where the function `is_foo()` is called but its return value is lost. This time, however, the unused return occurs alongside another symptom—a function that prints a message as well as returns a value. This may suggest confusion about the difference between printing and returning a value. As with the previous example, neither the symptom or the associated misconceptions will produce syntax or runtime errors but they may lead to unexpected output that is difficult to debug.

3 AUTOMATIC DETECTION OF MISCONCEPTIONS

Approaches to automatic detection of misconceptions include black box testing, comparison of novice solutions with expert solutions, interception of compiler errors, and custom parsing, which may be used in combination with each other and manual analysis.

Black box testing and comparison with expert solutions allow for the detection of problems beyond programming language misconceptions, including logic errors and strategic misconceptions e.g. [1, 11, 33]. However, both approaches require knowledge of the task the learner is working on, which is out of scope for our work.

An approach that does not require knowledge of the learner's task is to use error messages produced by the compiler e.g. [9, 11,

18, 19, 24]. This approach has several advantages: it is relatively straightforward to implement, error messages are consistent, pre-categorised, and contain information about the location and nature of the problem. However, error messages alone are often not sufficient to identify and understand student misconceptions as they do not always match the real cause of the error [9]. Another obvious disadvantage of the error message approach is that it will not catch symptoms that do not result in error messages, such as both examples in the previous section. For these reasons, error messages are often used as a starting point for misconception detection that is then supplemented with additional analysis.

Another approach is to use custom parsing to identify, for example, syntax patterns associated with errors, or changes in the source code over time. Where the aim of a project is to identify mistakes and misconceptions characterised by localised patterns independent of the larger program context, custom parsing has generally involved searching tokenised source code for clearly defined syntax patterns, e.g. use of = instead of == in a Boolean expression, or missing or incorrect use of punctation [5, 6, 9, 18]. Many of the papers that detect these types of mistakes provide little detail on the implementation of the parser; descriptions of the mistakes or misconceptions themselves typically provide enough detail to allow re-implementation by other researchers.

Detection of some misconceptions requires knowledge of the program structure as well as localised syntax patterns. For example, misunderstanding of variable scope may be indicated by the attempted use of a local variable in global scope. For these types of misconceptions, a parser that looks for patterns in tokenised source code will not provide enough information about the program structure. Some prior work has used the Abstract Syntax Tree of a program to enable easy parsing and identification of the various constructs and identifiers used as well as the relationships between them e.g. [16, 23, 31, 32].

For our own detection work, we use a custom parser with elements of an Abstract Syntax Tree. As we developed and tested the library, we were struck by the hidden complexity, challenging details, and large number of edge cases that had to be accounted for to implement reliable detection of many misconception symptoms, particularly those requiring understanding of the larger program context and how constructs such as conditionals and loops were used in combination. Many symptoms that we originally thought had simple and consistent forms turned out to be much more messy when tested with novice-written code that was unusually structured, creatively complicated, or in violation of conventions that more experienced programmers tend to follow without thought. Accounting for this messiness took a large amount of time and careful manual identification of edge cases that could easily be overlooked. The time factor is a primary motivation for the release of our library; we hope it will save other researchers investigating Python misconceptions a substantial amount of time. An additional motivation is that several design decisions made during development had noticeable impacts on the detection frequency of individual symptoms in our dataset. Due to the impact of detection details on results, particularly where quantification of misconceptions is a goal, we believe the detection process itself warrants greater attention than it has thus far received in misconception research using automated approaches.

4 DESIGN AND DEVELOPMENT OF SIDE-LIB

At the time of writing, SIDE-lib detects 33 symptoms and 25 potential misconceptions, with more to come. The full details of the detected symptoms and misconceptions are provided in the library[1].

4.1 Identification of Detectable Symptoms

We conducted a manual analysis of Python programs to identify symptoms of misconceptions that could be detected without knowledge of the programmer's task. Our dataset contains 1331 fully anonymised programs written by 123 graduate students in three different offerings of an introductory programming course. The first author was a co-instructor of the first offering and instructor of record for the second and third offerings, but was not involved in marking. Some programming prompts were used in multiple offerings, others were unique to one cohort.

Instead of using the full dataset to discover symptoms, we opted to select a sample of files for close inspection and save the rest for validation purposes. One team member made a pass through the full dataset to identify submissions that contained problems. We use the loose term "problem" to refer to either errors or misconceptions. This process involved selecting submissions that produced error messages, submissions that ran without error but produced different output than was described in the task instructions, or those that simply looked much longer or more complex than the sample solution. Programs with only task-specific problems (e.g. missed edge cases, output in a slightly different format than specified) were excluded. The resulting sample contained 186 programs.

We used collaborative coding [30] informed by existing inventories of errors and misconceptions (especially [1, 8, 15]) to develop a list of problems and their locations (file and line numbers) in the sample dataset. The resulting list contained 52 problems. The final step was to extrapolate symptoms from the problems and determine which could be automatically detected based purely on the code. We assumed that our detector would have access to variable names, variable scope and data types, and user defined function names, parameters, and return types. To identify detectable symptoms, we selected those that met the following criteria:

- The code that exhibits the problem is isolated—the problem remains apparent if the line or block of code where it occurs is viewed out of context of the rest of the file (except for the basic information described above e.g. variable scope).
- Instances of the problem have a consistent form, allowing for a small number of variations. This form represents the symptom.
- The problem can be identified without knowing the programmer's intention.
- The problem occurs more than once in the sample dataset.

Applying the criteria reduced the list of 52 problems to 33 detectable symptoms of problems. It is important to note that while some symptoms are always errors (e.g. an undefined variable), others are better described as code characteristics (e.g. a sequence of if statements without elif or else branches between) that will not be of interest unless other symptoms are also present.

[1] https://github.com/Supportive-IDE/SIDE-lib/

The symptoms were then mapped to potential misconceptions, a process that involved listing symptoms and other characteristics that had to be present in order to identify a misconception. So far, detection has been implemented for a total of 25 potential misconceptions that can be identified using the symptoms our library currently detects. 14 of the misconceptions are previously documented in [8] and [15], and 11 additional potential misconceptions came from our analysis of our dataset.

The list of symptoms detected by SIDE-lib covers most foundational programming concepts (variables, functions, conditionals, loops etc.) with one notable exception: classes. The files in our dataset do not include any examples of class definitions so misconceptions related to classes and object-oriented programming are omitted for now. We plan to add support for classes in future.

4.2 Implementation Details

Our detection library is implemented using JavaScript. Although JavaScript may seem like an odd choice for a Python parser, we felt that this would allow for easier use of the library in a wider range of scenarios than using the obvious development choice, Python. For example, our long term goals are to develop IDE extensions and browser-based tools for novices. Visual Studio Code, a very popular general purpose IDE is built with web technologies. Developing SIDE-lib with Python would have allowed us to take advantage of its built-in Abstract Syntax Tree capabilities but would complicate use of the library in other environments. Keeping the library entirely in JavaScript means that it can be included in web environments with one line of code. It can also easily be hosted and accessed as a web API.

The library entry point is a function that accepts Python source code and returns a JSON object containing information about the source code's "blocks", variables, functions, symptoms, and misconceptions. The information about blocks, variables, and functions is used internally to decide if and when symptoms indicate potential misconceptions. This supplemental information is provided to end users of the library to support additional analysis, including detecting misconceptions not currently supported by the library.

The detection process begins by tokenising the Python source code and determining the type and semantics of each token according to the language specification e.g. keyword, variable name, built-in function, assignment operator, literal value etc. During this process, information about the source code as a whole is collected in custom JSON objects representing the blocks, variables, and functions in the document.

The Python documentation defines a block as a piece of a program that is executed as a unit, specifically a module, class, or function [14]. We expand that definition and include any code that creates a branch, namely loops and conditionals. Information about blocks is stored as a tree, with each node including the type of block, the line number that it begins on, and references to child blocks.

Information about each variable includes its name, scope, whether or not it is a parameter, and a list of objects describing each usage of the variable—most importantly the line number and the data type at execution.

The collection of function information only includes details of user-defined functions in the parsed source code. Names and return types of built in functions and methods, as well as methods from some commonly used modules, are stored in lookup tables. For each user-defined function, we track its name, where in the document it is defined, the names of any parameters, information about any returns (data type, location), and where in the document the function is called, if applicable.

Knowledge of expression data type is needed for the detection of 6 of the symptoms and 7 of the misconceptions implemented so far. Determining data types in Python can be complex because the language allows data types of variables to change, compound types such as lists can store a mix of data types, and functions may return different data types from different branches. Where an expression is guaranteed to have a single data type, such as a variable assigned a literal value or the result of a built-in function with a single possible return type, the expression is assigned that data type. Otherwise, if the expression could have more than one possible data type, or there is not enough information to determine type, it is labelled "unknown".

The data types of multi-part expressions such as calculations and concatenations are evaluated but, if any part of the expression has unknown data type, the resulting data type of the compound expression will also be labelled unknown. Operations involving unknown data types are assumed to be appropriate to avoid detecting a problem when none is present. An expression will only be identified as having invalid data type if it is the result of an operation involving incompatible known data types, such as adding a String to an integer. Although our library recognises compound data types like lists, it does not track the data types of items inside compound data types—these items are considered to have unknown data type. Additionally, parameters are always labelled as having unknown data type at definition, even if the intended data type can be inferred from subsequent operations or values passed as arguments when a function is called. This is because the possibility still remains that the function may be passed a different data type, e.g., if the function is called by unit tests in a separate file.

Once parsing is complete, symptoms are identified according to rules unique to each symptom. For the simplest symptoms, it is a case of looking for clearly defined patterns in sequences of tokens. Other symptoms involve cross referencing token patterns with the supplemental program information.

Each instance of a symptom is stored in an object containing the symptom name, its location in the document, and the affected snippet of source code. Some symptoms also include additional fields providing more information. For example, an instance of an unused return value (see Figure 1) includes the name of the function that was called and whether it is built-in or user defined. The information provided about instances of symptoms helps in the identification of potential misconceptions but it is also intended to facilitate the development of learner (or teacher) facing systems that provide feedback about symptoms.

The final step in the detection process is to use the symptoms to identify potential misconceptions. For each misconception, we have defined rules about which symptoms should be present and, if multiple symptoms are involved, how they relate to each other.

These rules are similar to the notion of a constraint in constraint-based modelling, which has been used to identify errors and provide personalised feedback in intelligent tutoring systems [27].

A symptom can appear multiple times in a single program but a potential misconception will only be recorded once. Information about each detected misconception is stored in a JSON object that contains its name, and a list of symptom occurrences that explain why the potential misconception has been identified. Each occurrence stores information about the location in the file, the specific symptom instances that caused the misconception to be identified, and a text explanation that is customised to the symptom instances involved (see Figure 2). For some misconceptions, there are multiple combinations of different symptoms that could lead to its detection. One example is *PrintSameAsReturn*—possible confusion around printing a value versus returning a value from a function. There are two combinations of symptoms that may indicate this misconception:

- The result of a call to a function that does not return a value is assigned to a variable or otherwise used in an operation (*AssignedNoReturn*) AND either the same function contains print statements (*FunctionPrints*) OR the function involved is the built in `print` function. See Figure 2.
- The result of a call to a function that returns a value is not assigned to a variable or otherwise used in an operation (*UnusedReturn*) AND the same function contains print statements (*FunctionPrints*). See Figure 1, line 11.

The information provided in each occurrence in the JSON returned by the library allows for more fine grained analysis of how and why the misconception has been identified.

4.3 Validation

Validation of SIDE-lib's symptom detection process was performed by testing library output for our dataset against the results of our analysis during the design process (Section 4.1) to match detected symptoms. We manually checked and fixed all discrepancies.

To validate the misconceptions, we manually checked that all identified misconceptions matched the results of our analysis, including the contributing symptoms of each misconception. It is important to note that ground truth for misconceptions is less concrete than for symptoms as we can't be sure if the cause behind the symptoms we identify is really the misconception we think it is. Many of the misconceptions in our own work and much (though certainly not all) of the prior research categorising errors and misconceptions is based on the insights of educators who work with learners, rather than the direct words of the learners themselves. Therefore, our validation was really a case of checking that potential misconceptions are identified according to the rules that we have derived from prior work and our own experiences.

To guard against the influence of factors specific to the courses in which our dataset was collected, such as style and structure conventions, or the manner in which particular concepts were taught, we also manually inspected the output of the library on programs in a dataset collected in a different setting [31]. The two datasets are qualitatively quite different. Where our dataset is made up of homework submissions, featuring complete programs of varying complexity, the dataset from Rivers et al. contains in progress solutions to much smaller exercises. This means that we have been able to verify that our detector is robust against common characteristics of incomplete code such as mismatched parentheses and half finished statements.

```
1  def get_direction(start, end):
2    if start < end:
3      print("Southbound")
4    elif start > end:
5      print("Northbound")
6    else:
7      start == end
8      print("Invalid")
9
10 direction = get_direction(8, 7)
```

```
  contributingSymptoms: [
    {
      line: 9,
      type: "AssignedNoReturn",
      text: "get_direction",
      docIndex: 198,
      expressionNoValue: {
        type: "userDefinedFunction",
        value: "get_direction"
      }
    },
    {
      line: 0,
      type: "FunctionPrints",
      text: "get_direction",
      docIndex: 4,
      printLines: [2, 4, 7],
      functionReturns: false
    }
  ],
  explanation: "User-defined function
      get_direction prints to the console
      but does not return a value."
```

Figure 2: A program (top) containing evidence of the misconception *PrintSameAsReturn*. The function beginning on line 1 exhibits the symptom *FunctionPrints*, which is not an issue unless other symptoms are present. Line 10 has symptom *AssignedNoReturn*. Together, these symptoms indicate the potential misconception. The bottom part of the figure shows a segment of the JSON object returned for this occurrence of the *PrintSameAsReturn* misconception. The `contributingSymptoms` and `explanation` fields provide information about why the misconception has been detected.

5 EXAMPLE APPLICATIONS

We have designed SIDE-lib to be useful for a range of use cases beyond our own long term goals. In this section, we present two example applications built using the library. The first is an analysis

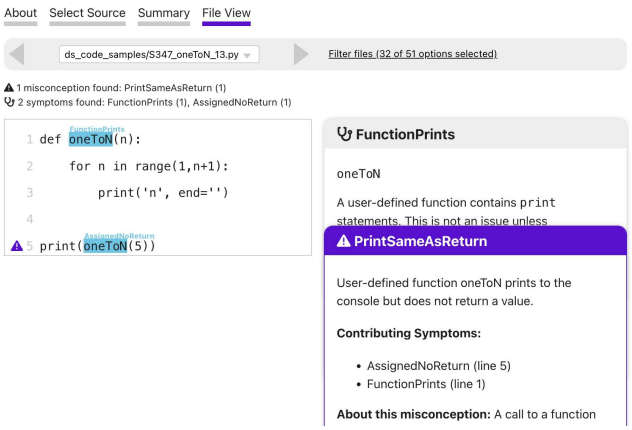

Figure 3: The File view visualises detected symptoms and misconceptions for a single file.

tool aimed at researchers that we are using in our own work. The second application is an early-stage prototype of basic IDE-based feedback for learners when a misconception is detected.

5.1 The Symptom Checker Website

The Symptom Checker website[2] allows users to upload a dataset of Python files and view the detected symptoms and misconceptions. When upload is complete, each file is passed to SIDE-lib for parsing. The user is then taken to the Summary view, which shows frequency tables of symptoms and misconceptions detected by SIDE-lib for the whole dataset.

The File View section of the website (Figure 3) visualises symptoms and misconceptions detected in a single file. Files can be filtered by symptom and misconception. The location of each symptom is highlighted and labelled in the source code and more information is provided in the cards to the right of the code. Clicking a particular misconception highlights the specific symptom instance(s) that contributed. The Symptom Checker was developed to simplify our process of identifying and validating symptoms and misconceptions by visualising SIDE-lib's output in a human friendly format. All information displayed, including the in-code highlights of symptoms is created using the data retrieved from the JSON object output by SIDE-lib.

5.2 A Visual Studio Code Extension

Another use case is IDE-based support for novice programmers. As SIDE-lib is written in JavaScript, it is straightforward to include in browser-based IDEs and desktop applications developed with web technologies.

Figure 4 shows a prototype of a basic Visual Studio Code extension that sends the contents of the code editor to SIDE-lib every time there is a change. If misconceptions are detected, a short action-oriented message is displayed directly above the affected code. In the example in Figure 4, a file from the Rivers et al. dataset [31], the misconception *MapToBooleanWithIf* [8] is detected due to the form of the conditional beginning on line 3. As this is not a critical

[2]https://supportive-ide.github.io/symptom-checker/

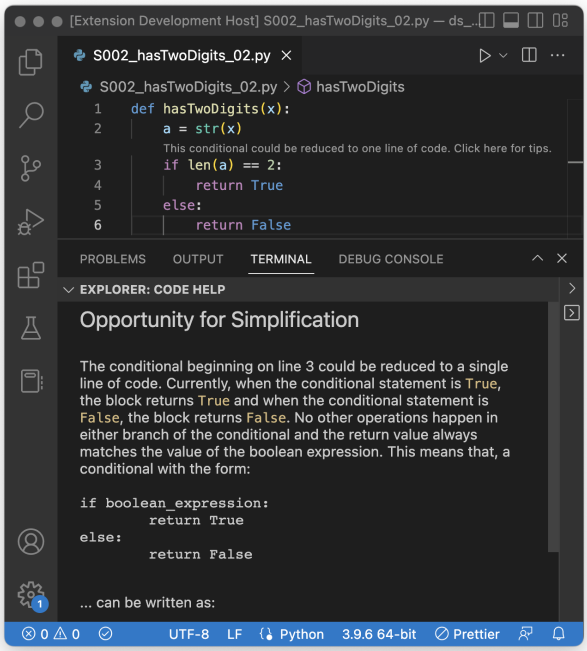

Figure 4: A Visual Studio Code extension that notifies users of detected misconceptions and provides basic feedback.

misconception, the message hints at an opportunity for improvement and invites the user to click on it for tips. When the user clicks the message, it opens a custom web view—the "Code Help" area shown in the bottom half of the screen in Figure 4. The web view provides feedback and guidance. The messaging and feedback is created independently of the SIDE-lib allowing for a range of interactive interventions beyond the display of static information shown in this example.

6 CONCLUSION

This paper has described the design and development of SIDE-lib, a library that detects task-independent symptoms of potential misconceptions in Python code. Although we are far from the first to implement automatic detection of programming misconceptions, our contribution is making our detector available to other researchers as a flexible standalone tool. We found that implementation of the detection, particularly for symptoms of misconceptions that go beyond localised syntax patterns, was extremely time-consuming. We hope that by releasing our library, we can save other researchers time and reduce effort required to re-implement detection of misconceptions across projects.

REFERENCES

[1] Ella Albrecht and Jens Grabowski. 2020. Sometimes It's Just Sloppiness - Studying Students' Programming Errors and Misconceptions. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*. ACM, Portland OR USA, 340–345. https://doi.org/10.1145/3328778.3366862

[2] Amjad Altadmri and Neil C. C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large Scale Student Data. In *Proceedings of*

the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '15). ACM, Kansas City MO USA, 522–527. https://doi.org/10.1145/2676723.2677258

[3] Piraye Bayman and Richard E. Mayer. 1983. A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements. *Commun. ACM* 54, 9 (Sept. 1983), 677–679. https://doi.org/10.1145/358172.358408

[4] Piraya Bayman and Richard E. Mayer. 1988. Using Conceptual Models to Teach BASIC Computer Programming. *Journal of Educational Psychology* 80, 3 (1988), 291–298.

[5] Neil C. C. Brown and Amad Altadmri. 2014. Investigating Novice Programming Mistakes: Educator Beliefs vs. Student Data. In *Proceedings of the 10th Annual Conference on International Computing Education Research (ICER '14)*.

[6] Neil C. C. Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A Large Scale Repository of Novice Programmers' Activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, Atlanta GA USA, 223–228. https://doi.org/10.1145/2538862.2538924

[7] Ricardo Caceffo, Pablo Frank-Bolton, Renan Souza, and Rodolfo Azevedo. 2019. Identifying and Validating Java Misconceptions Toward a CS1 Concept Inventory. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM, Aberdeen Scotland UK, 23–29. https://doi.org/10.1145/3304221.3319771

[8] Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. 2021. A Curated Inventory of Programming Language Misconceptions. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '21)*. ACM, Virtual Event, 380–386. https://doi.org/10.1145/3430665.3456343

[9] Thomas Dy and Ma Mercedes Rodrigo. 2010. A Detector for Non-Literal Java Errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling, '10)*. ACM, Berlin, Germany, 118–122. https://doi.org/10.1145/1930464.1930485

[10] Anna Eckerdal and Michael Thune. 2005. Novice Java programmers' conceptions of 'object' and 'class', and variation theory. In *Proceedings of the 2005 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '05)*. ACM, Monte de Caparica Portugal. https://doi.org/10.1145/1067445.1067473

[11] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common Logic Errors Made By Novice Programmers. In *Proceedings of the 20th Australasian Computing Education Conference (ACE '18)*. ACM, Brisbane Australia, 83–89. https://doi.org/10.1145/3160489.3160493

[12] Ann E. Fleury. 1991. Parameter Passing: The Rules Students Construct. In *Proceedings of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education (SIGCSE '91)*. ACM, San Antonio TX USA, 283–286. https://doi.org/10.1145/107004.107066

[13] Ann E. Fleury. 2000. Programming in Java: Student-Constructed Rules. In *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education (SIGCSE '00)*. ACM, Austin TX USA, 197–201. https://doi.org/10.1145/330908.331854

[14] Python Software Foundation. 2023. Execution Model. Retrieved Jan 17, 2023 from https://docs.python.org/3/referenceexecutionmodel.html

[15] Guilherme Gama, Ricardo Caceffo, Renan Souza, Raysa Benatti, Tales Aparecida, Islene Garcia, and Rodolfo Azevedo. 2018. *An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in Python.* Technical Report IC-18-19. Institute of Computing, University of Campinas, Campinas Brazil.

[16] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. 2018. Misconception-Driven Feedback: Results from an Experimental Study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*. ACM, Espoo Finland, 160–168. https://doi.org/10.1145/3230977.3231002

[17] Simon Holland, Robert Griffiths, and Mark Woodman. 1997. Avoiding Object Misconceptions. In *Proceedings of the 28th ACM Technical Symposium on Computer Science Education (SIGCSE '97)*. ACM, CA USA, 131–134. https://doi.org/10.1145/268084.268132

[18] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. In *Proceedings of the 34th ACM Technical Symposium on Computer Science Education (SIGCSE '03)*. ACM, Reno NV USA, 153–156. https://doi.org/10.1145/611892.611956

[19] Matthew C Jadud. 2005. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education* 15, 1 (2005), 25–40. https://doi.org/10.1080/08993400500056530

[20] Fionnuala Johnson, Stephen McQuistin, and John O'Donnell. 2020. Analysis of Student Misconceptions using Python as an Introductory Programming Language. In *Proceedings of the 4th Conference on Computing Education Practice 2020 (CEP 2020)*. ACM, Durham UK, 1–4. https://doi.org/10.1145/3372356.3372360

[21] Fionnuala Johnson, Stephen McQuistin, John O'Donnell, and Quintin Cutts. 2022. Experience Report: Identifying Unexpected Programming Misconceptions with a Computer Systems Approach. In *Proceedings of the 2022 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '22)*. ACM, Dublin Ireland. https://doi.org/10.1145/3502718.3524775

[22] Lisa Kaczmarczyk, Elizabeth R. Petrick, Philip J. East, and Geoffry L. Herman. 2010. Identifying Student Misconceptions of Programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*. ACM, Milwaukee WI USA, 107–111. https://doi.org/10.1145/1734263.1734299

[23] David Liu and Andrew Petersen. 2019. Static Analyses in Python Programming Courses. In *Proceedings of the Fiftieth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, Minneapolis MN USA, 666–671. https://doi.org/10.1145/3287324.3287503

[24] Davin McCall and Michael Kölling. 2014. Meaningful Categorisation of Novice Programming Errors. In *Proceedings of the 2014 Frontiers in Education Conference (FIE '14)*. Madrid Spain. https://doi.org/10.1109/FIE.2014.7044420

[25] Davin McCall and Michael Kölling. 2019. A New Look at Novice Programmer Errors. *ACM Transactions on Computing Education* 9, 4 (2019), 1–30. https://doi.org/10.1145/3335814

[26] Tanya J. McGill and Simone E. Volet. 1997. A Conceptual Framework for Analyzing Students' Knowledge of Programming. *Journal of Research on Computing in Education* 29, 3 (1997), 276–297. https://doi.org/10.1080/08886504.1997.10782199

[27] Antonija Mitrovic. 2012. Fifteen Years of Constraint-Based Tutors: What We Have Achieved And Where We Are Going. *User Modelling and User-Adapted Interaction* 1-2 (2012), 39–72. https://doi.org/10.1007/s11257-011-9105-9

[28] Roy D. Pea. 1986. Language-Independent Conceptual 'Bugs' in Novice Programming. *Journal of Educational Computing Research* 2, 1 (Feb. 1986). https://doi.org/10.2190/689T-1R2A-X4W4-29J2

[29] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education* 18, 1 (2017), 24. https://doi.org/10.1145/3077618

[30] K. Andrew R. Richards and Michael A. Hemphill. 2018. A Practical Guide to Collaborative Qualitative Data Analysis. *The Journal of Teaching in Physical Education* 37, 2 (2018), 225–231. https://doi.org/10.1123/jtpe.2017-0084

[31] Kelly Rivers, Erik Harpstead, and Ken Koedinger. 2016. Learning Curve Analysis for Programming: Which Concepts do Students Struggle With?. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. ACM, Melbourne Australia, 143–151. https://doi.org/10.1145/2960310.2960333

[32] Kelly Rivers and Kenneth R. Koedinger. 2015. Data-Driven Hint Generation in Vast Solution Spaces: A Self-Improving Python Tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2015), 37–64. https://doi.org/10.1007/s40593-015-0070-z

[33] Teemu Sirkiä and Juha Sorva. 2012. Exploring Programming Misconceptions: An Analysis of Student Mistakes in Visual Program Simulation Exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli Calling '12)*. ACM, Koli Finland, 19–28. https://doi.org/10.1145/2401796.2401799

[34] Juha Sorva. 2012. *Visual Program Simulation in Introductory Programming Education.* Ph. D. Dissertation. Aalto University, Espoo Finland.