Learnersourcing at Scale to Overcome Expert Blind Spots for Introductory Programming: A Three-Year Deployment Study on the Python Tutor Website

Philip J. Guo
UC San Diego
La Jolla, CA, USA
pg@ucsd.edu

Julia M. Markel
UC San Diego
La Jolla, CA, USA
jmarkel@ucsd.edu

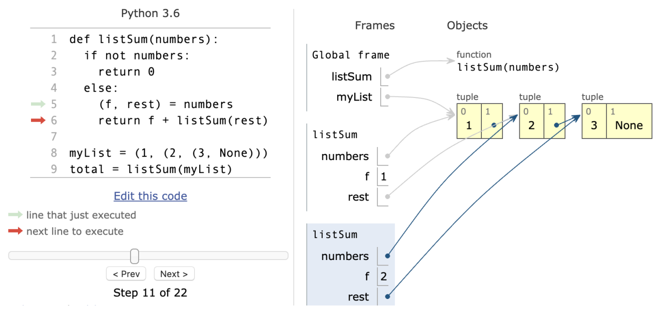
Xiong Zhang
University of Rochester
Rochester, NY, USA
xzhang92@cs.rochester.edu

ABSTRACT
It is hard for experts to create good instructional resources due to a phenomenon known as the *expert blind spot*: They forget what it was like to be a novice, so they cannot pinpoint exactly where novices commonly struggle and how to best phrase their explanations. To help overcome these expert blind spots for computer programming topics, we created a learnersourcing system that elicits explanations of misconceptions directly from learners while they are coding. We have deployed this system for the past three years to the widely-used Python Tutor coding website (`pythontutor.com`) and collected 16,791 learner-written explanations. To our knowledge, this is the largest dataset of explanations for programming misconceptions. By inspecting this dataset, we found surprising insights that we did not originally think of due to our own expert blind spots as programming instructors. We are now using these insights to improve compiler and run-time error messages to explain common novice misconceptions.

INTRODUCTION
Novices suffer from a large variety of misconceptions when learning computer programming, ranging from misunderstandings about syntax to incorrect mental models of code execution [5]. Although ideally they would have human tutors to help them, in practice millions of people are now learning online from self-paced tutorials, YouTube videos, and MOOCs where they do not have convenient access to human experts.

One way experts can scale their efforts is to write explanations for novice misconceptions and display them inline within instructional resources. But a fundamental shortcoming of this approach is that experts often suffer from a cognitive phenomenon called the *expert blind spot* [4], also known as the *curse of knowledge*: They forget what it was like to be a novice, so they have a hard time empathizing with the struggles of novices. In the context of learning programming:

L@S '20, August 12–14, 2020, Virtual Event, USA.
© 2020 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7951-9/20/08 ...$15.00.
http://dx.doi.org/10.1145/3386527.3406733


```
def listSum(numbers):
  if not numbers:
    return 0
  else:
    (f, rest) = numbers
    return f + listSum(rest)

myList = (1, (2, (3, None)))
total = listSum(myList)
```

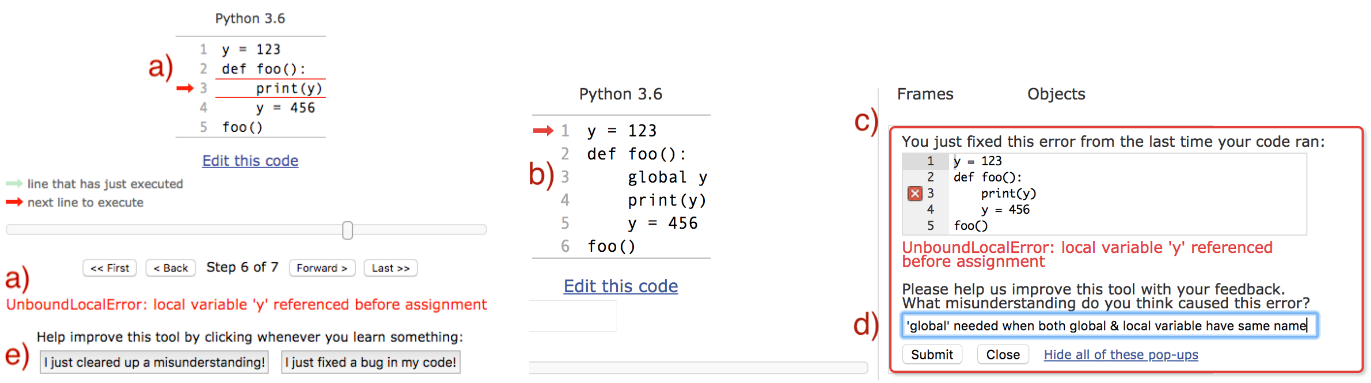
Figure 1. Python Tutor [2] is a website that lets users write code (left) and see how it executes step-by-step with visualizations of run-time state (right). In this work-in-progress, we augment Python Tutor with learnersourcing to collect novice misconceptions about programming errors.

- Experts may forget all the places in the code where novices commonly struggle, since for them everything looks simple.
- Experts may provide incomplete explanations since they assume novices have more prior knowledge than they do.
- Experts may use advanced jargon and vocabulary in their explanations, since they assume novices know those terms.

Our hypothesis in this work-in-progress is that ***novices can help overcome expert blind spots by providing their own written explanations for common coding errors that they are facing***. Specifically, novices know exactly where they struggle since they experience it firsthand; and once they overcome those struggles they can hopefully write explanations using terminology that fellow novices can relate better to.

To explore this hypothesis at scale, we took a learnersourcing approach [6, 7] where we built a novel system to collect crowdsourced explanations of programming misconceptions by using learners as the crowd. We deployed this system within Python Tutor [2], an online code editor and visual debugger with tens of thousands of daily active users and over ten million total users so far (Figure 1). Our intuition is that by prompting learners there to provide explanations *at the exact moment they overcome a particular coding struggle*, we can collect a corpus of written explanations that can augment educational materials to overcome common expert blind spots.

We deployed this system online for three years (2017–2020) and collected 16,791 learner-written explanations spanning a variety of novice coding misconceptions. By inspecting these explanations, we (as experienced programming instructors)



Python 3.6

```
1  y = 123
2  def foo():
3      print(y)
4      y = 456
5  foo()
```

UnboundLocalError: local variable 'y' referenced before assignment

Help improve this tool by clicking whenever you learn something:

I just cleared up a misunderstanding!
I just fixed a bug in my code!

Python 3.6

```
1  y = 123
2  def foo():
3      global y
4      print(y)
5      y = 456
6  foo()
```

You just fixed this error from the last time your code ran:

```
1  y = 123
2  def foo():
3      print(y)
4      y = 456
5  foo()
```

UnboundLocalError: local variable 'y' referenced before assignment

Please help us improve this tool with your feedback.
What misunderstanding do you think caused this error?

'global' needed when both global & local variable have same name

Figure 2. a) The user encounters an error while coding in Python Tutor, b) fixes their code and re-runs, c) sees a pop-up box showing the error they just fixed, and d) writes an explanation about what misconception led them to make that error. e) The user can also write freeform explanations at any time.

have found surprising insights that we did not originally think of due to our own expert blind spots. We can use these insights to augment tools with more novice-friendly explanations.

The research contributions of this work-in-progress are:

- A novel and scalable learnersourcing technique for collecting novice explanations of programming misconceptions.
- Preliminary findings from a three-year deployment that collected a dataset of 16,791 learner-generated explanations.

RELATED WORK

Learnersourcing is a crowdsourcing technique where learners contribute annotations for future learners while they are using an instructional resource [6]. It has been applied to generating subgoal labels for educational videos [6], debugging hints for computer engineering coursework [1], and explanations for math problem solutions [7]. *To our knowledge, we are the first to use learnersourcing to collect novice misconceptions about computer programming.* Prior work has catalogued such misconceptions from the instructor's perspective, but they have not attempted to collect explanations directly from learners [5]. The closest system to ours is HelpMeOut [3], which aggregates Java errors into a web interface that an expert instructor can later annotate with their notes; in contrast, our system directly queries learners in-situ and was deployed for three years to generate a large-scale dataset of learner explanations.

LEARNERSOURCING SYSTEM PROTOTYPE

Figure 2 shows our system prototype embedded within the Python Tutor website: a) The user can directly write code on the site or copy-paste in code examples and problems they are working on from online tutorials. They will inevitably encounter errors when they are coding, which are either syntax errors or run-time errors. Figure 2a shows a run-time error when line 3 is executed, with Python reporting "UnboundLocalError: local variable 'y' referenced before assignment." These message are hard for novices to understand [5] since they use technical jargon and do not indicate *why* someone might have made that error (i.e., what their misconception was). Syntax errors in Python show even less helpful messages, most commonly "SyntaxError: invalid syntax." b) After they eventually fix the error, here by adding '`global y`' on line 3, they run the code again and the error is gone. c) At that moment, our system pops up a dialog showing the user's previous code, its error message, and the question "What misunderstanding do you think caused this error?" d) The user can write a response or dismiss the dialog. Hopefully they write an explanation using terms that fellow learners can empathize with better than Python's built-in error messages; in this example, the user wrote "'global' needed when both global & local variable have same name." Their text is saved to our corpus, along with the history of their coding session, which includes both the erroneous and fixed code. Our rationale for designing the system this way is that we want to collect the learner's thoughts ***right at the moment when they just fixed an error*** so that their misconception is at the top of their mind.

Our prototype automatically pops up a dialog whenever the user fixes a syntax or run-time error. Note that since it only detects that an error message has disappeared, it is possible to get false positives when the user, say, completely changes their code in between attempted executions. In that case, the original error may be gone, but they simply moved on without trying to fix it. In the future, we could heuristically suppress false positives by taking diffs between code executions and not popping up a dialog box if the code diff is too large.

This prototype can collect explanations for a variety of errors that novices encounter, which include both compile-time (syntax) and run-time errors that Python automatically flags. However, it is possible for code to execute to completion without Python issuing any errors, but it still produces incorrect results. These are known as semantic or logic errors [5], and they are impossible for a system to detect without a test suite. Since Python Tutor users mostly write freeform code without a test suite[1], we wanted to provide a way to collect learnersourced explanations for semantic errors. Thus, we added two buttons to the bottom of the interface, shown in Figure 2e. The user can click either "I just cleared up a misunderstanding!" or "I just fixed a bug in my code!" at any time, which then prompts for a written explanation. These buttons give users a way to share *freeform explanations* for moments when they get a sudden insight about what is wrong with their code, even when Python cannot detect a syntax or run-time error.

[1]If a test suite were available, then the system could automatically pop up a dialog whenever a test goes from failing to passing, since that may indicate that a semantic error was fixed.

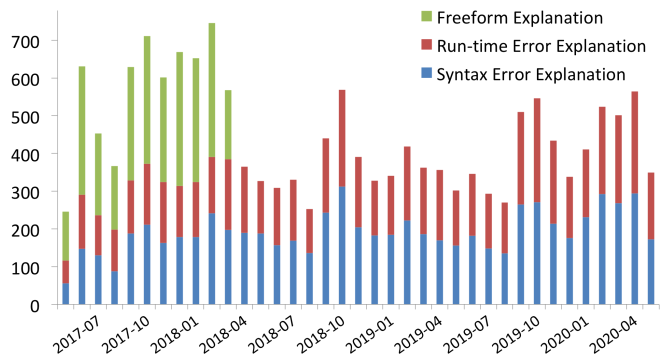


Figure 3. Number of learner-submitted explanations per month (May 2017–May 2020) for syntax errors (blue), run-time errors (red), and freeform explanations using the buttons in Figure 2e (green). Due to the host website's UI changes, freeform was discontinued in March 2018.

We designed this system to be lightweight and unobtrusive so that it can be deployed live to the Python Tutor website [2] without adversely affecting learners' experiences. It triggers only when errors have been fixed or when the user clicks the buttons in Figure 2e. The user can also choose to never see these pop-ups; this may cause us to miss out on some potential data, but it respects the rights of users not to be disturbed.

PRELIMINARY FINDINGS

We deployed this system to the Python Tutor website in 2017, so we have collected over three years of data on its usage. For this preliminary analysis, we took a three-year corpus of data from May 19, 2017 to May 19, 2020. We also only looked at data for Python 3, which is the most common language that learners write on the Python Tutor site. (Other supported languages include Python 2, Java, JavaScript, C, and C++.)

We explored two main research questions using this data:

1. Would learners be willing to use this system voluntarily as they are working on their code on the Python Tutor website?
2. Can these explanations potentially help overcome expert blind spots [4] and improve future instructional materials?

Learners submitted 16,791 explanations that were at least 10 characters long; we filtered out short strings since they were uninformative. Submissions came from 141 countries, with the most from the U.S., India, Canada, U.K., and Australia. (Nearly all explanations were in English.) Out of these 16,791 explanations, 7,466 were in response to fixing syntax errors, 6,333 for fixing run-time errors, and 2,992 were freeform explanations using the buttons in Figure 2e. Figure 3 shows an average of around 300 submissions per month, or 10 per day.

This level of usage suggests that learners were willing to voluntarily contribute explanations, although response rates were low. During this time period, the syntax error pop-up dialog box was shown 1,906,878 times but collected only 7,466 explanations: a 0.4% response rate. Run-time errors had a 0.3% response rate. This was unsurprising since we did not give learners any incentive (other than altruism) to contribute; and even those who contributed were unlikely to submit an explanation for *every* error they saw. Most of the time they just fixed the error and moved on with their coding task.

Nonetheless, we collected a large enough corpus to discover many insights that surprised us as experienced programming instructors. *To our knowledge, this is the largest corpus of explanations for programming misconceptions.* In this preliminary analysis, we grouped entries by error type and skimmed all of the explanations to find common patterns.[2]

Many errors were trivial (e.g., typos, mismatched parentheses), so here we report only those where learner-written explanations gave meaningful insights about their misconceptions.

Linguistic Misconceptions

Many misconceptions involve troubles with mapping between the syntax of natural languages (e.g., English) and code syntax. Our own expert blind spots caused us to never think about many of these since we had been programming for so long that code syntax came "naturally" to us. But learners wrote insightful explanations for many kinds of errors, including:

- Omitting quotes for strings: In natural language, there is no need to use quotes for prose. Omitting a single-word string like `foo(Alice)` is legal code but accesses an undefined variable; omitting a multi-word string like '`foo(My name is Alice)`' leads to a parse error. Escape sequences like \" are also confusing since they are not needed in English.
- Capitalization: Some learners capitalize the first letter of variable/function names when they appear at the start of a line of code, because in English the first word in a sentence should always be capitalized. Those same names are *not* capitalized when used later, which causes errors since most programming languages are case-sensitive. They also sometimes mistakenly capitalized keywords such as `For`, `While`, `If`, again because it looks more natural in written English.
- Singular/plural: A common idiom for iteration is to use a plural noun for a collection and singular for each element in it, such as '`for name in names:  <loop body>`'. Some learners were confused about whether to access elements inside the for-loop using singular or plural. Singular `name` is correct most of the time, but sometimes the entire collection needs to be accessed, such as `names.remove()`.
- Pronoun references: Some learners used "pronouns" to refer to their variables with shorthand, which surprised us. For instance, in many small pieces of learner code, there is only one object that is created and being operated on, such as a list. After defining the list using a variable name, some learners simply referred to that list as `list` later in their code instead of using its name, which throws an error.
- Verb placement: Function calls are like verbs, so some learners were confused by where to place them in a statement (akin to a "sentence" in code). For instance, some wrote `parse(x)=input()` since it reads left-to-right like *"parse the string x that comes from user input"* but the correct syntax is `x=parse(input())` if `parse` returns a string.
- Incorrectly chaining conditions, like '`if x != a or b`' which reads naturally in English as *"if x is not equal to a or b."* We also saw cases like '`if x > min and < max`' which reads as *"if x is greater than min and less than max."*

[2]Showing them to other instructors would be better for ecological validity, but for this work-in-progress we started with ourselves.

- Iteration that reads like English: Variables like `i` are often used to iterate from 0 to some upper bound. Some learners wrote code like '`while i <= 100`' or '`for i in 100`', which read in English like `i` will iterate up to 100. The former fails because `i` is never initialized, and the latter fails because for-loops need a collection to iterate over.

Mathematical Misconceptions

Since many intro. programming problems deal with numbers and math, we discovered some surprising misconceptions from mapping between the syntax of math and code. For instance:

- Upon learning that == means *equals* in programming, some learners tried using `x == y` as an assignment statement instead of `x=y`, since they wanted to "make x equal to y." This intuitively makes sense but has no side effect in code (it just returns a boolean). These errors are hard to track down since only later in execution when `x` is accessed will they realize that it does not have the updated value from `y`.
- Similarly, some did not know that assignment statements go from right to left like `c=a+b`, so they wrote left-to-right assignments like `a+b=c`. The unhelpful error here is "can't assign to operator." Again, based on mathematical equality, order does not matter, but in programming it does matter.
- Omitting * for multiplication: Instead of writing `3*x`, they write `3x`, which gives a syntax error since it looks like a variable name that starts with a digit. `x3` does not work either since it also denotes a variable name. Also, in math `a(b+c)` and `(a+b)c` both mean multiplication, but in Python the former signifies a function call of `a()` and the latter is a syntax error. Note that Python may display vastly different error messages for the same misconception, and those errors give no indication of the learner's underlying math confusion.
- When numerical data is read from files or terminal input, they often start as strings. If they are not properly converted to numbers, it is still possible to use math operators like + and > on them, which will perform string concatenation and comparison, respectively. These can lead to subtle semantic and logic errors, even though the code does not crash.

Polyglot Programming Misconceptions

Many learners came to Python from other languages like Java, C, C++, or JavaScript. As polyglot programmers, they wrote explanations for some common cross-language errors:

- API mismatches: Many languages share identically-named functions for operating on built-in types. Some learners were surprised that Python's API differed in subtle ways, such as the string `split()` method not accepting some kinds of parameter values that are allowed in JavaScript.
- Compiled languages allow code to call a function (or class) that is defined syntactically lower down in the source file, but in Python everything must be defined before being used.
- Python uses colon+indentation instead of {braces} for block scopes, which was a common frustration. More subtly, other languages allow single-line blocks without braces, but in Python it still needs colon and, optionally, newline+indent; this fact led to surprising and non-obvious error messages.

DISCUSSION AND ONGOING WORK

This preliminary analysis revealed some of our own expert blind spots as programming instructors. While many misconceptions seem apparent in retrospect, we never even considered many of them before reading these learner-submitted explanations. Some of these have been reported in prior studies mostly from classroom settings [5], but the novel contributions of our scalable technique and large data set are: 1) We can automatically collect data that tells us exactly which Python error messages map to specific misconceptions, and how frequently those occur in the wild. 2) We logged all of the code that led to those errors, which we can later distill into error-inducing code snippets. 3) We have explanations written in the learners' own words, which can help us create custom error messages using terminology that learners can better relate to.

Using this data, we are developing an automated tool that can detect the most common types of learner misconceptions and give more helpful error messages than what Python prints by default. Developing this tool is mostly a matter of engineering effort in writing custom parsers, heuristics, and analyzing run-time value data for run-time errors. However, we would not know what features we should include in such a tool if not for our data-driven approach. Without learnersourcing, we would need to manually sift through millions of automatically-logged syntax and run-time errors and *guess* what those learners intended to do instead of reading their firsthand explanations.

That said, learner-provided explanations have some limitations: As expected, many are low-quality and even inaccurate, so the signal-to-noise ratio is low. It still takes many hours of our time to read through thousands of these explanations to find patterns. In the future, we could augment learnersourcing with a system where Python Tutor users vote on which explanations make the most sense to iteratively improve their contents.

Acknowledgments: This material is based upon work supported by the National Science Foundation under Grant No. NSF IIS-1845900.

REFERENCES

[1] Elena L. Glassman, Aaron Lin, Carrie J. Cai, and Robert C. Miller. 2016. Learnersourcing Personalized Hints *(CSCW '16)*. ACM.

[2] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education *(SIGCSE '13)*. ACM.

[3] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What Would Other Programmers Do: Suggesting Solutions to Error Messages *(CHI '10)*. ACM, 1019–1028.

[4] Mitchell J Nathan, Kenneth R Koedinger, and Martha W Alibali. 2001. Expert blind spot: When content knowledge eclipses pedagogical content knowledge. In *Proceedings of the third international conference on cognitive science*. Beijing: University of Science and Technology of China Press, 644–648.

[5] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (Oct. 2017).

[6] Sarah Weir, Juho Kim, Krzysztof Z. Gajos, and Robert C. Miller. 2015. Learnersourcing Subgoal Labels for How-to Videos *(CSCW '15)*. ACM.

[7] Joseph Jay Williams, Juho Kim, Anna Rafferty, Samuel Maldonado, Krzysztof Z. Gajos, Walter S. Lasecki, and Neil Heffernan. 2016. AXIS: Generating Explanations at Scale with Learnersourcing and Machine Learning *(L@S '16)*. ACM, 379–388.