Supplement for Scope Rule Comprehension by Novice Python Programmers

Mark A. Holliday

Department of Mathematics and Computer Science

Western Carolina University

Cullowhee, NC USA

holliday@wcu.edu, ORCID 0009-0001-2700-1957

Abstract

This document is a supplement to the extended abstract presented at the poster session of SIGCSE TS 2025. It goes into more details about the rule set that describes the Python scope rule features covered by the assessment.

Understanding scope rules, the rules for the binding of variable uses with variable declarations, is a necessary skill for reading and understanding program code. It is also one that novice programmers often find challenging. To better understand which misconceptions novice Python programmers commonly hold about scope rules, we pose the question of "To what extent do misconceptions exist even for a simple subset of scope rule features?". The answer to this question can determine the level of complexity of scope rule features the instructor should focus on.

We consider a subset of Python that only has non-recursive and non-nested function definitions, scalar variables, and global code that calls the function definitions. We also consider the variable categorization into local variables, parameters, and global variables.
We developed an assessment tool that we used as part of the final examination in a CS1 course for three semesters. We found that though there were significant misconceptions for some features, the number of such features is limited. This identification of the common misconceptions for this subset of Python can aid in revising the instructional focus to help future students better understand scope rules.

Section 1. Introduction

There is a combination of features in Python and similar languages involving scope rules, aliases, parameter passing, and the related effects of mutation. Multiple studies indicate that novice programmers often have difficulty with these concepts resulting in incorrect mental models and misconceptions[fleury91, fisler17, kohn23,stromback23]. We chose a subset of features for our experiment that addresses only

- the binding of variable uses to variable declarations; and
- the classification of those variable declarations as global variables, parameters, and local variables (since the name binding rules depend on the kind of variable)

In addition, nested functions, references, recursion, and class definitions are not included. By solely focusing on name binding, mutation of values is not addressed.

More precisely our notional machine addresses the following features of a program.

- When is an occurrence of a variable name a *declaration* of a new variable or a *use* of an existing variable?
- How is it determined to which, if any, variable declaration, a particular variable use binds?
- How do we determine what kind of variable (*global variable*, *parameter*, or *local variable*) a particular variable use is using?

In Section 2 we review related work about notional machines and scope rules. In Section 3 we introduce the notional machine, its visual representation, and describe which Python features are not included but are related and could occur in more complex notional machines that build upon the simple notional machine presented here. Section 4 introduces the assessment tool that we have developed to evaluate how well the students' mental models match the notional machine. The assessment tool is introduced by showing how it has been used as a final examination question in three semesters. Section 4 also includes our observations on the results of those final examination questions. We conclude in Section 5.

Section 2. Related Work

In learning to program there is a set of concepts that are often described together and include scope rules, aliasing, parameter passing and the affect that mutation has on these features. A number of papers have investigated how well students, usually novice programmers, comprehend these concepts. Fleury[fleury91] examined student understanding of parameter passing at the end of an introductory, college-level Pascal programming course. Six small programs were used each to check understanding with three of the programs overlapping with our notional machine. The students struggled on those three programs with the "vast majority" having incorrect answers for one program and about half incorrect (at least initially) for the other two programs.

Fisler[fisler17] conducted pre-tests and post-tests of upper-level computer science majors in Java and Scheme in an advanced programming language course. Most of their questions address concepts beyond our notional machine, but the paper shows two questions one in Scheme and one in Java, that address concepts in our notional machine. At least 80% of the class correctly answered questions about basic scopes in Scheme in both the pre-quiz and post-quiz but did not do as well in Java. The one Scheme question shown only involves some of the concepts covered by our notional machine, so it is not possible to determine the extent to which their questions cover all the scope rules equivalent to the Python ones we examine.

Clements[clements22] investigates student understanding of program features such as threads, callbacks, and recursion which are beyond the scope of our notional machine and which require the stack to be an essential part of the notional machine.

Strömbäck[stromback23] compared student understanding of scope, parameter passing, and aliasing for computer science students during their first and second years and, for some students, their third year. Three of the fifteen questions in the C++ program version answered by the students do overlap with our notional machine. For those three questions many of the students in their second and third years gave incorrect answers (including 74% incorrect for one of the questions).

Kohn[kohn23] like us examines how novice programmers understand scopes in six small Python programs. Of the related studies, the notional machine implicit in their questions most resembles ours. The programs used do not test some features of our notional machine such as the `global` directive, attempts to use a variable before its declaration, and the classification of variables as global, parameters, and local variables upon both declarations and uses. The university students performed relatively

well on their three questions with at most 16% incorrect numneric answers on any one of the three questions. The high school students did not do as well on their three questions with 70%, 23%, and 47% incorrect numeric answers. It would be interesting to compare how well the students would do using the assessment tool described in this paper instead of providing numeric answers.

The simplicity of the set of scope rule related features we consider is a key difference from our experiment and these previous experiments.

Section 3. Features specified by a rule set

Subsection 3.1. Features

Our thesis is that addressing only the binding of variable uses to variable declarations as well as identifying whether a variable is a global variable, parameter, or local variable and only using assignments of numbers, strings, and booleans may be helpful with student understanding as well as teacher assessment of student misconceptions. This occurs before we introduce related concepts involving mutation, references, aliasing, and data structures.

Consequently, we view a Python program code fragment as:

- a set of variable name occurrences with each occurrence either a declaration, a use, or neither a declaration of a valid use;
- each variable name occurrence that is a use binds to one and only one declaration of that variable;
- each variable name occurrence that is not a declaration or a use does not bind to any declaration of that variable; and
- each variable name occurrence that is valid (that is, a declaration or a use) is either a global variable, a parameter, or a local variable. The kind of variable is determined by the location of its declaration, not the location of a use.

Line Number	**Name for the Variable**	**Is this a variable declaration?**	**If it is not a variable declaration, is it a valid use?**	**If it is a valid use, on what line is the declaration that is being used?**	**If it is a declaration or a valid use, what kind of Variable (Local Variable, Parameter, Global Variable)?**	**Rule***
1	style	Yes	--	--	param	param R1
3	style	No	Yes	1	param	param R2
4	color	No	Yes	11	global	directive R1
5	tone	No	No	--	--	localVar R2
6	tone	Yes	--	--	local var	localVar R1
9	style	Yes	--	--	local var	localVarR1
11	color	Yes	--	--	global	global R1
12	color	No	Yes	11	global	global R3

Caption: Assessment Tool Solution for Spring 2023 and Spring 2024 Code in Figure 7. The Rule column is not part of the assessment tool.

Subsection 3.2. Rule Sets and Visual Representation

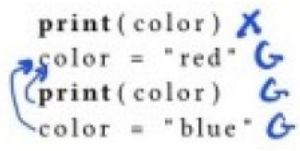
The semantics of the notional machine are organized into a series of six rule sets which are labelled and listed below. Unlike the **rules of program behavior** of Duran et al.[duran21] these rules are student-facing as well as teacher-facing. Above each rule set is a code fragment illustrating the visual representation of that rule set.

Our visual representation is simple:

- the kind of variable (**G** for global, **P** for parameter, and **LV** for local variable) appears next to each variable occurrence;
- if a variable occurrence is not a declaration or use, an X is shown; and
- if a variable occurrence is a use then an arrow is shown to the declaration to which that use is bound

Subsubsection 3.2.1. Rule Set 1 named `global`

1. Rule 1. The first global assignment to a variable is a declaration of a global variable.
2. Rule 2. A global variable cannot be used before it is declared.
3. Rule 3. If a variable global occurrence appears after a global declaration of that variable, then that variable occurrence is a valid use of that global variable.



Caption: Rule Set 1 named **global** Rule set with scope rules for global variables.

Subsubsection 3.2.2. Rule Set 2 named `param`

1. Rule 1. An occurrence of a variable in the parameter list of a function is a declaration of a parameter variable.
2. Rule 2. Any occurrence of a variable in the function body of a variable declared as a parameter is a valid use of that parameter.

```
def go(color):
  print(color)
  color = "red"
```

Caption: Rule Set 2 named **param**. Rule set with scope rules about parameters.

Subsubsection 3.2.3. Rule Set 3 named `localVar`

1. Rule 1. If there is no `global` directive, then the first assignment statement with a given variable on its left hand side is a declaration of a local variable.
2. Rule 2. Any occurrence of a variable in the function body before the variable declaration statement is not a valid use or declaration.
3. Rule 3. Any occurrence of a variable in the function body after the variable declaration statement is a valid use.

```
def go():
  print(color)
  color = "red"
  print(color)
  color = "blue"
```

Caption: Rule Set 3 named **localVar**. Rule set with scope rules for variables declared in a function body.

Subsubsection 3.2.4. Rule Set 4 named `directive`

1. Rule 1. If there is a `global` directive for a variable in a function body, then any occurrence of that variable in that function body is a valid use of the global variable if the global variable is declared before the function call.

```
def go():
  global color
  color = "red"
  print(color)
color = "blue"
go()
```

Caption: Rule Set 4 named **directive** Rule set with scope rules involving the `global` directive.

Subsubsection 3.2.5. Rule Set 5: named `fncOnly`

1. Rule 1. A variable declared as a parameter or local variable in one function is not visible outside of that function body.

```
def go(color):
  score = 5

def go1():
  print(color)
  print(score)

print(color)
print(score)
go()
go1()
```

Caption: Rule Set 5 named **fncOnly**. Rule set with scope rules for declarations within a function only being visible within that function.

Subsubsection 3.2.6. Rule Set 6: named `LGB`

1. Rule 1. If there is no `global` directive in a function body and there is no declaration of the variable in the function body, then the binding of any occurrence of that variable uses the LGB (Local-Global-Builtin) rule.
2. Rule 2. If there is a `global` directive in a function body and there is a declaration of that variable in the global namespace before the call to the function, that declaration is used.
3. Rule 3. If there is a `global` directive and there is a declaration of that variable in the global namespace but it is after the call to the function, then the variable occurrence in the function body is not a valid use.

```
def go():
  print(color)
  print(score)

color = "red"
go()
score = 5
```

Caption: Rule Set 6 named **LGB**. Rule set with scope rules using the \textit{local-global-builtin} sequence.

Subsection 3.3. Python Features Not Included

Our approach defines a notional machine as simply as possible to aid in the construction of accurate mental models by the students which are then expanded by incrementally adding more Python features via additional notional machines. Several Python features we are not including are listed below.

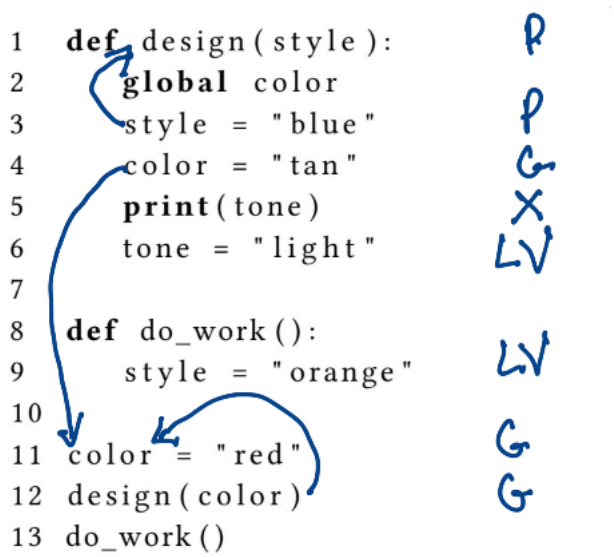
1. *references*. Data structures such as lists require adding the concepts of references which adds a significant complexity to the mental model the students are constructing.
2. *nested functions*. Nested functions require adding *non-local variables* as a kind of variable and some relatively straightforward rules. Including nested functions would not require a major change in the notional machine, but does add some unnecessary complexity.
3. *recursive functions*. Recursive functions require a substantially different notional machine. With recursive functions there is no longer a one-to-one mapping between a variable declaration and a line of code. A single line of code may declare multiple variables with each variable associated with a different recursive call of the function. Clements et al.\cite{clements22} is a study of aspects of scope rule comprehension that uses a notional machine and takes into account recursive functions. Consequently, the program stack is an integral part of their notional machine.
4. *class definitions*. Class definitions require the introduction of a new kind of variable, *data attributes*, with their own distinctive scope rules. This would add unnecessary complexity

Whether a variable occurrence in a function body is a valid use of a variable declared globally depends on whether the call to the function is before or after the variable declaration. This implies that if in the global code there are two calls to that function, one before the global variable declaration and one after, then sometimes an occurrence of the variable in the function being called is a valid use and sometimes it is not.

Due to the static nature of our notional machine we would handle this ambiguity by listing both cases for a given variable occurrence. This was too subtle for our students at this point in their learning. As a result, in all of our code examples and code questions using our notional machine we always show the global declaration of a variable before any call to a function that includes a variable use binding to that variable declaration.

Line Number	Name for the Variable	Is this a variable declaration?	If it is not a variable declaration, is it a valid use?	If it is a valid use, on what line is the declaration that is being used?	If it is a declaration or a valid use, what kind of Variable (Local Variable, Parameter, Global Variable)?	Rule*
1	color	Yes	--	--	param	param R1
2	color	No	Yes	1	param	param R2
3	letter	No	Yes	9	global	LGB R1
4	sound	Yes	--	--	local var	localVar R1
6	color	Yes	--	--	param	param R1
7	sound	No	No	--	--	fncOnly R1
9	letter	Yes	--	--	global	global R1
11	letter	No	Yes	9	global	global R1

Caption: Assessment Tool Solution for Fall 2023 Code in Figure 8. The Rule column is not part of the assessment tool.



Caption: Examination Code for Spring 2023 and Spring 2024.

Section 4. Assessment Tool

As support for our notional machine we present an assessment tool which appears to be useful in determining student misconceptions of name binding and could be beneficial in aiding students in recognizing their misconceptions. A use of the assessment tool includes a small Python program and a blank table that includes a row for each occurrence of a variable name. In a row there are columns for:

- whether the variable occurrence is a declaration;
- if not a declaration, whether it is a valid use; and
- if it is a valid use which line has the variable declaration
- the kind of variable (global, parameter, or local variable)

For this paper the table includes an additional column identifying the rule being used. This column is not part of the table presented to the students.

We use this tool in our lecture notes, as an in-class exercise and also as an examination question. Given the code listing, the students fill in the table which is originally empty. In some cases no variable declaration is visible to a variable occurrence. We do not treat the `global` directive as a declaration or use.

Subsection 4.1. Spring 2023 and Spring 2024 Examinations

In this subsection we illustrate the assessment tool by the question we used in our Spring 2023 and Spring 2024 final examination with the solution shown in Table 1.

Subsection 4.2. Fall 2023 Examination

In this subsection we illustrate the assessment tool by the question we used in our Fall 2023 final examination with the solution shown in Table 2.

Caption: Examination Code for Fall 2023.

Rule Set	**Rule**	**Spring 2023 (21 students)**			**Fall 2023 (19 students)**	
		Rules	**Kind of Var**	**Total**	**Rules**	**Kind of Var**
global						
	Rule 1	5%	10%	14%	11%	0%
	Rule 2	na	na	na	na	na
	Rule 3	29%	24%	52%	16%	11%
param						
	Rule 1	24%	0	24%	0%	0%
	Rule 2	43%	10%	52%	21%	21%
localVar						
	Rule 1	0%	10%	10%	5%	0%
	Rule 2	0%	0%	0%	na	na
	Rule 3	na	na	na	na	na
directive						
	Rule 1	43%	5%	48%	na	na
funcOnly						
	Rule 1	0%	10%	10%	16%	0%
LGB						
	Rule 1	na	na	na	58%	11%
	Rule 2	na	na	na	na	na

Caption: Percentage of students with incorrect responses for each rule for each semester.

Subsection 5.2. Observations

We did not include tests of **global R2** and **localVar R3** because we tested the very similar **localVar R2** and **global R3**, respectively. We also did not test for **LGB R2** since we had only briefly discussed it in class. It is important to note that by the time of the final examination in each of the semesters the students have seen and used this notional machine and assessment tool several times including in the lecture, an in-class exercise, and at least one previous test. Our university's institutional review board approved our study protocol. We anonymized the data by using only the page of the examination paper with the question and answer, but no student name or other personal information.

Table 3 shows the number of students in each course and the percentage of students who made a mistake involving each of the rules of the notional machines. Sometimes students correctly used a rule but incorrectly identified the kind of variable. Table 3 divides the mistakes into the case of a rule misapplication and the case of a correct rule application but incorrect variable classification.

The **global R3** rule provides that an occurrence after a global declaration is a valid use and is a basic property. In a substantial number of the mistaken test answers the rule is applied correctly but the kind of variable is misidentified as a parameter or local variable. That the variable name in question also occurs in a function body may be a factor in the misconception. That the kind of variable is determined by the location of the declaration, not the location of the use, may still be not understood by some students.

The **param R2** rule provides that an occurrence in the same function of a parameter is a use of that parameter. This also is a basic property similar to the **global R3** rule. Again, a high percentage of answers show a misconception. In two of the three semesters the primary problem involved that the rule is being misapplied instead of the misconception being in the variable classification. In the variable classification errors typically the variable is misidentified as a local variable.

The **directive R1** rule provides that the existence of the \textbf{global} directive causes the occurrences of the variable in that function body to refer to a global variable declaration. This was the third rule that produced a high percentage of incorrect answers and indicates a lack of understanding of the meaning of the `global` directive.

The **LGB R1** rule also yielded a high percentage of errors and the errors occur primarily in the application of the rule itself, not in variable classification. This is a relatively difficult concept so the high percentage of misconceptions
involving the rule itself instead of variable classification is not as surprising.

Subsection 5.3. Summary

In summary there are areas that the students often exhibited misconceptions but they are relatively few and specific. They are:

- that variable classification into local variable, parameter, and global variable is based on the location of the declaration, not the location of the use
- if a function has a parameter, any occurrence of that parameter's name in that function's body is a use of that parameter
- the `global` directive.
- a requirement for a global variable to be visible to a variable use in a function body the declaration of the global variable has to be before the call to the function

Section 6. Conclusions

We developed an assessment tool to help understand the misconceptions of novice Python programmers that involve scope rules. Our focus is on a simple subset of Python features related to scope rules. Thus, our understanding the extent of misconceptions for these simple features is not hindered by misconceptions of more complex features of Python that are related to scope rules.

We used our assessment tool in three final examinations of a CS1. There are substantial errors of the students' answers in some cases, but the areas of misconceptions exhibited are relatively few and specific. It is quite possible that with some adjustment or extension of instruction in those areas these misconceptions can be resolved. The focus can then on identifying the misconceptions involving more complex features of Python scope rules.

References

[clements22] John Clements and Shriram Krishnamurthi, *Towards a Notional Machine for Runtime Stacks and Scope: When Stacks Don't Stack Up*, "2022 ACM Conference on International Computing Education Research", "ICER 2022", "pages 206-222", "Lugano, Switzerland", "August 2022", {ACM}, {New York, NY, USA}, https://doi.org/10.1145/3501385.3543961.

[fisler17] Kathi Fisler, Shriram Krishnamurthi, and Preston Tunnell Wilson, *Assessing and teaching scope, mutation, and aliasing in upper-level undergraduates*, "Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education", "SIGCSE 2017", "pages 213--218", "Seattle, WA, USA", "2017", {ACM, New York, NY, USA}, http://dx.doi.org/10.1145/3017680.3017777.

[fleury91] Ann E. Fleury, *Parameter passing: The rules that students construct*, "Proceedings of the 22nd SIGCSE Technical Symposium on Computer Science Education", "SIGCSE 1991", "pages 283-386", "New York, NY, USA", "1991", {ACM, New York, NY, USA}, http://dx.doi.org/10.1145/107004.107066.

[kohn23] Tobias Kohn and Dennis Komm, *Coping with Scoping: Understanding Scope and Parameters*, "28th Annual Conference on Innovation and Technology in Computer Science Education", ITiCSE 2023, "July 2023", "pages 201--207", {ACM, New York, NY, USA}, https://doi.org/10.1145/3587102.3588798.

[stromback23] Filip Strömbäck, Pontus Haglund, Aseel Berglund, and Erik Berglund, *The Progression of Students' Ability to Work With Scope, Parameter Passing and Alaising*, "Proceedings of the 25th Austalasian Computing Education Conference (ACE '23)", "ACE 2023", "pages 39-48", "Melbourne, VIC, Australia", "February 2023", {ACM, New York, NY, USA}, https://doi.org/10.1145/3576123.3576128.