April, 2019

Learning to Get Literal: Investigating Reference-Point Difficulties in Novice Programming

Craig S. Miller, *DePaul University*
Amber Settle

Learning to Get Literal: Investigating Reference-Point Difficulties in Novice Programming

CRAIG S. MILLER, DePaul University
AMBER SETTLE, DePaul University

We investigate conditions in which novices make some reference errors when programming. We asked students from introductory programming courses to perform a simple code-writing task that required constructing references to objects and their attributes. By experimentally manipulating the nature of the attributes in the tasks, from identifying attributes (e.g. *title* or *label*) to descriptive attributes (e.g. *calories* or *texture*), the study revealed the relative frequencies with which students mistakenly omit the name of an identifying attribute while attempting to reference its value. We explain how these reference-point shifts are consistent with the use of metonymy, a form of figurative expression in human communication. Our analysis also reveals how the presentation of examples can affect the construction of the reference in the student's solution. We discuss plausible accounts of the reference-point errors and how they may inform a model of reference construction. We suggest that reference-point errors may be the result of well practiced habits of communication rather than misconceptions of the task or what the computer can do.

CCS Concepts: •**Applied computing** → *Education;*

Additional Key Words and Phrases: metonymy; novice computer programming; reference errors; human-computer interaction; misconceptions

ACM Reference format:
Craig S. Miller and Amber Settle. 2019. Learning to Get Literal: Investigating Reference-Point Difficulties in Novice Programming. *ACM Trans. Comput. Educ.* x, x, Article x (x 2019), 17 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Human listeners effortlessly comprehend many forms of figurative expression. For example, consider the sentence: "Open the ice cream and serve two scoops." Most people readily infer that the speaker is asking for a *container* to be opened, not the ice cream itself. In this case, the figurative expression is an instance of metonymy, where the speaker indicates an item (ice cream) that is related to the intended reference (container).

Metonymy has been extensively studied in the context of human language [14, 24]. While there is some debate on the mechanistic properties that underlie metonymy [19, 28], the process arguably draws upon knowledge of the domain in resolving the metonymic reference [8]. Returning to the ice cream example, the listener needs to know that ice cream comes in a container, which then must be opened in order to serve the ice cream. As in this case, the use of metonymy often facilitates understanding by simplifying the wording (*open the ice cream* vs. *open the container of ice cream*) and emphasizing the identifying element (e.g. serve ice cream instead of some other dessert).

DOI: 10.1145/nnnnnnn.nnnnnnn

In contrast to the relative ease with which humans comprehend figurative language such as metonymy, it presents difficulties with human-to-machine communication, particularly in the domain of programming. A previous study has shown how novice programmers mistakenly refer to a whole object when they intend to reference an attribute belonging to the object and vice versa [18]. Consistent with the use of metonymy, the incorrect shift in reference more frequently occurs among identifying attributes, such as a *title* attribute, than among descriptive attributes, such as *color*. As an example, when the programming task calls for writing the title attribute of an object (e.g. **obj**), a novice programmer may incorrectly omit the attribute and produce the following code:

```
write(obj)
```

In contrast, if the programming task calls for writing the color attribute of an object, a novice programmer is more likely to correctly include the attribute with the reference:

```
write(obj.color)
```

Other computing education researchers have studied student errors in the context of human communication and language. In many cases, students mistakenly draw upon the everyday use of a term when using it in a programming context [4, 7, 21, 33]. More generally Tenenberg and Kolikant [37] discuss how practices of human communication influence student understanding of how computers interpret code. Finally, instructors have acknowledged student difficulty in constructing precise expressions and have created classroom exercises where students practice giving clear instructions to a human acting as a very literal robot [9, 15].

While previous study of novice programming has linked linguistic expression to student errors, previous research has yet to systematically study any particular phenomenon in order to work out theoretical underpinnings and offer effective instructional strategies. Here we focus on reference-point errors and explore how communication practices of figurative expression may underlie the errors. Effective instruction would then promote practices that eschew the figurative, metonymic expression and aim for the literal construction required of human-to-computer communication. In the next sections, we review previous theoretical work before we consider a study that systematically explores the conditions under which reference-point errors occur.

1.1 Theoretical basis of reference-point errors

One theoretical analysis of reference-point errors [19] has outlined three possible knowledge sources for why novice programmers produce reference-point errors that are consistent with the use of metonymy:

- **Deficient mental representation**. A novice programmer's mental representation of the object may be lacking an explicit mention of any identifying attribute. For example an identifying attribute, such as its name, may have a privileged status. Such status makes its name (e.g. apple) interchangeable with the object itself when referencing either.
- **Misunderstanding of what the system can do.** A novice programmer may have an incorrect understanding of the notional machine, that is, how the computer interprets and processes the given code [10, 32]. In this case, a student may assume that the notional machine can infer the programmer's intention without explicit reference to the object's attribute.
- **Reliance on implicit habits for communication**. Correct knowledge for specifying the complete reference may be missing or inaccessible, in which case, the novice programmer relies on knowledge acquired and practiced in human communication, where metonymic references are routinely used.

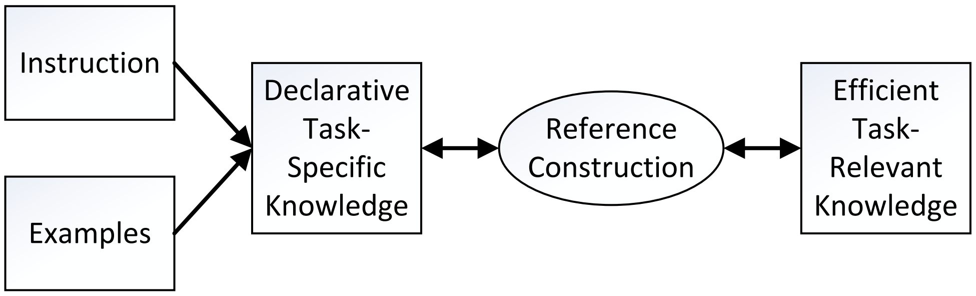
Instruction
Examples
Declarative Task-Specific Knowledge
Reference Construction
Efficient Task-Relevant Knowledge

Fig. 1. Knowledge sources for problem solving.

An alternate, yet complementary, approach for analyzing reference-point errors is to consider the knowledge *systems* that novice programmers employ as they construct a reference. Figure 1 depicts potential knowledge sources and systems for applying them as students construct references when programming. It draws upon the theoretical framework presented by Pirolli and Recker [25], where problem solutions are constructed from two different systems of knowledge application. Efficient task-relevant knowledge is acquired from well-practiced prior experiences; it is fast and generally not accessible for verbal presentation. In contrast, declarative task-specific knowledge draws upon deliberate, interpretive processing, which includes instruction and examples; it is slow and amenable to verbal reflection. As Pirolli and Recker note, this theoretical framework is the basis of well-established theories of human cognition and, in particular, the ACT* cognitive architecture [1].[1]

For reference construction, unpracticed knowledge originating from examples and instruction fits the description of the declarative knowledge system. On the other hand, if the novice programmer is drawing upon habits of communication such as metonymy, these well-practiced habits are described by the efficient, procedural knowledge system.

A reference-point error that arises from practiced habits may be similar to a Stroop effect [34]. Using one variant of a Stroop task, studies have shown that people are slower and more error-prone in naming the color appearance (e.g. red) of a word when the word spells an incongruent color (e.g. ‘blue’) than when the word spells a congruent color (e.g. ‘red’). Prominent theories account for the difficulty by acknowledging that the practiced habit of reading interferes with the task [16]. Similar to the Stroop task, practiced habits of communication may give novice programmers difficulty when constructing a reference.

Even though the Stroop task may engage different perceptual and cognitive mechanisms than reference construction, it provides a useful device for discussing whether a misconception underlies the reference-point errors observed with novice programmers. Misconceptions are often defined as a mistaken belief about a concept, the computing system (i.e. notational machine) or the computing task (see Qian and Lehman [27] for a discussion on the diverse definitions). Misconceptions have been frequently used to explain student difficulties [2, 7, 12, 31, 36]. As we have already discussed, reference-point errors may arise from a misconception of how the object is represented or from a misconception of what the notional machine can do. In contrast, if practiced habits of communication are interfering with correct reference construction, the error is similar to a

[1]Kahneman [13] provides a broadly accessible overview of these two systems of thinking.

Stroop-effect error, which are not misconceptions of the task but rather skill-based slips. The distinction is important because it might call for different instructional interventions. Instruction and examples may be sufficient to correct misconceptions. However, correcting skill-based errors might require teaching students strategies for noticing the error and practicing how to override any interference.

1.2 Previous Study and Goals of this Study

We have already presented a study on reference construction difficulties in a broader programming context [20]. The purpose of that study was to understand the implications of the transparency of the Python programming language when defining objects. We found that students had difficulty aligning parameter references when constructing method definitions. Also noted, but not fully explored, was the frequency of reference errors to objects. We found that reference-point errors were a problem for participants and were more common in complex expressions than in simple ones. Part of the experiment involved varying the type of attribute needed. While identifying attributes were more frequently omitted, we suspect that our sample size was too small to yield a statistically reliable result.

Here we add to our previous work [20] by drawing upon a larger sample, which allows us to focus on factors that underlie the reference-point errors that we previously observed. As we will see, our analysis provides stronger evidence that students are more likely to omit identifying attributes than descriptive attributes. It also replicates similar results from previous studies [17, 18], while demonstrating that these errors occur over a range of attributes, student populations, and programming tasks.

An additional contribution of this current paper is our examination of how instructional examples affect the construction of references by students. While not the focus of the original study, we will see how the successful construction of references can be traced back to particular examples used in the instructional text. Prior research has theorized how novice programmers draw upon examples, often internalized as partial schemas or plans, for resolving problem-solving impasses [11, 25, 29]. For our analysis, we will draw upon the model that Pirolli and Recker [25] applied to account for novice skill acquisition. Their study also demonstrates benefits of reflection and self-explanation for learning, which we will consider when recommending effective instructional strategies.

Ultimately, given this paper's focus on how students construct code-based references, we aim to 1) identify and confirm when reference-point errors occur, 2) suggest possible instructional interventions to support student learning, and 3) provide some insights on how the observed reference errors fit with a theoretical framework adopted from Pirolli and Recker.

2 STUDY OF REFERENCE ERRORS

As we have already noted, we originally conducted our study to explore the transparent elements of Python's object-oriented mechanisms on student learning [20]. This paper describes the same study and method but its findings draw upon a larger sample of participants. In our work we ask students to write a new method to an already defined class (presented in the appendix). This programming task involves writing a "lookup" routine, whose solution requires a sequential search, an equality test that accesses object attributes, and a return statement that also references an object attribute.

The solution requires attribute references in the equality comparison and the return statement. As already noted, previous research [12, 18] indicates that students are more likely to construct an incorrect reference involving attributes with identifying labels (e.g. *name*, *id*, *title*) than with descriptive labels (e.g. *color*, *texture*, *shape*). For example, a student is more likely to just reference an object (e.g. **obj**) when the intention is to reference an identifying attribute that belongs to the object (e.g. **obj.title**).[2] To verify this effect, the study randomly selects attributes from a set of identifying and descriptive attributes for the programming task. By analyzing the relative frequencies by which students correctly reference the attribute, we can test whether the type of attribute is a determining factor for reference-point errors.

3 METHOD

3.1 Participants

We recruited 80 students from two first-year programming courses. One course is a second-quarter Python course in a sequence designed for novice programmers. The other course is an accelerated Python course for students who have already taken at least one programming course. More information about the topics and pacing for the two courses is provided in Section 7.1.

3.2 Programming Task

Students were presented a programming task through an online web application. The online presentation provided students with two class definitions: an **Item** class and a **Catalog** class. The exercise then instructed students to add a simple "lookup" method to the Catalog class.

The Item class implements simple objects with attributes and values (e.g. name: 'apple', texture: 'smooth') and the Catalog class represents a list of these items and includes methods for adding an item and obtaining the number of items in the list. The Catalog class also provides a method (has_style), whose structure is similar to the method that they are asked to write (lookup). The code for the Item and Catalog classes is given in the appendix.

The lookup method involves searching the catalog list for an object with a particular attribute and then returning another specified attribute for the matching object. The complete instructions for the programming task are presented in the next section.

The web application systematically varied the targeted attribute and returned attribute. The set of attributes (e.g. *name*, *texture*) were chosen to invoke a grocery domain and used example values accordingly (e.g. *apple*, *smooth*). The complete set of possible attributes are calories, color, label, name, shape, status and texture. While the web application randomly selected without replacement all pairs of attributes, an accounting error in the software produced nearly twice the number of tasks where the name was the targeted attribute and calories were the returned attribute.

Below is a correct solution for the **lookup** method given **shape** as the targeted attribute and **name** as the returned attribute.

```
def lookup(self, target):
    'returns needed attribute'
    for obj in self.items:
        if obj.shape == target:
            return obj.name
    return None
```

3.3 Problem Wording for Exercise

Below is an example set of instructions that were presented to students for the programming problem. This particular set asked the student to find the targeted *shape* attribute and return the *name* attribute of the matching object.

[2]Like many object-based programming languages, Python uses dot-attribute notation to reference an attribute that belongs to an object. For example, **obj.color** refers to the color value of the obj object. Specifying just the object (e.g. **obj**) refers to the whole object.

Exercise Problem

The following is a short programming problem. Since we are interested in learning about initial strategies and difficulties, we ask that you do not try to run your code when providing your solution. After you submit your untested code, you will see a correct solution and will be able to compare it to your code.

For this problem, Item and Catalog classes have been defined.

The code for both the Item and the Catalog class can be viewed in **this page**. *Note: 'This page' links to the code presented in the appendix.*

The Item class creates objects consisting of attributes (e.g. "style") and values (e.g. "flat"). When a new Item object is created, an id attribute is automatically assigned. Here's an example of interacting with the class in a console:

```
>>> obj = Item()
>>> obj.id
102
>>> obj.style = "flat"
>>> obj.style
'flat'
>>>
```

Note that Python allows attributes to be assigned values with an assignment statement. In addition to id and style, possible attributes for the Item class include calories, color, label, name, shape, status and texture.

The Catalog class produces objects that contain a collection of items. Here's an example of interacting with the Item and Catalog class:

```
>>> listing = Catalog()
>>> listing.add(obj)
>>> listing.has_style("flat")
True
>>> listing.has_style("twisted")
False
>>> listing.size()
1
```

Problem: Write a method for the Catalog class called **lookup**. This method should take a string as an argument to find the first object that matches its shape attribute. The method should then return the name of the matching Item object.

For example, consider that listing is a Catalog object that has an Item with shape of 'round' and name of 'apple'. Then, this call:

```
listing.lookup('round')
```

would return

```
'apple'
```

If there is no matching object, the method should return the Python keyword **None**.

Using just the editing space below, write your definition of the **lookup** method so that it works as described above. When you have finished, click on the submit button below the editor.

3.4 Procedure

Students were provided with a URL for the pages containing the code written by the authors and the programming task instructions. Students who wished to participate completed the consent form on the first page. A second page collected demographic information including age, gender, and the number of prior programming courses. A subsequent page presented the exercise instructions and related code. This page included an embedded editor with syntax-aware formatting for Python. Students used this editor to submit their definition of the lookup method. The online application recorded the time each student took to read the instructions to the programming task and submit their solution. After submitting their solution they were provided with the correct solution and asked to comment on their submission.

For our previous study we recruited 36 participants [20]. For this work we recruited additional participants to produce a total of 80 submissions. Our analysis suggested that a larger sample would produce statistically reliable differences.

4 RESULTS

4.1 Demographics

Of the 80 students who participated in the study, the median reported age was 20 (mean = 22.2) with 14 students identifying as female and 66 as male. The median reported number of prior programming courses was 2 (mean = 2.3).

4.2 Coding

For our analysis, we focus on the expressions contained in the *if* comparison and the return statement. If the required attribute (e.g. *name*, *color*, *id*) was explicitly coded in the expression, the expression was categorized as an *explicit* reference. We were able to employ a computer script that automatically coded the student responses according to the examples and principles presented below.

Here is one student answer that correctly references the *color* attribute to find the matching object and then literally references the *status* attribute in the return value:

```
def lookup(self, color):
    for obj in self.items:
        if obj.color == color:
            return obj.status
```

Here is an answer with some incorrect programming syntax yet it (correctly) employs an explicit reference to the *label* attribute in the if comparison and to the *shape* attribute in the return statement:

```
def lookup(self, label):
    if self.label() == label:
        return self.shape()
```

If the expression contained no reference to the attribute, the expression was categorized as an *implicit* reference.

When determining whether the attribute is explicitly referenced, parameter references were excluded from consideration. This exclusion was operationally accomplished by removing the first appearance of any reference that appears in the parameter list. For example, this student answer has the variable *color* appearing in the parameter list:

```
def lookup(self, color):
    for obj in self.items:
        if color in self.items:
            return obj.texture
    return None
```

In this case, our analysis removed the *color* attribute from the expression in the *if* statement. Since the expression has no other explicit reference to color, the if expression was coded as having an implicit reference. The *return* statement for this answer does contain an explicit reference to the required attribute (i.e. **obj.texture**); consequently this expression was coded accordingly.

Student responses that did not contain an *if* statement or a return statement were coded as missing for their respective expression.

Finally, we observed that many of the student responses explicitly referenced the *style* attribute in both statements. This was particularly common when the problem required a reference to the *texture* attribute. This occurred in 7 out of 14 texture cases, such as in the following student answer, which asked for the texture of the object to be returned:

```
def lookup(n):
    for obj in self.items:
        if obj.calories == n:
            return obj.style
        else:
            return None
```

In these cases, the explicit reference to the style attribute was coded as an explicit response.

4.3 Analysis

Table 1 presents the proportions of attributes that were explicitly referenced, broken down by whether the expression appeared in the if statement or in the return statement. The table in the second column also indicates the example value (e.g. red for color) that was presented in the instructions. As can be inferred from the table, the 80 student participants yielded 73 coded responses from the if comparison and 74 coded responses from the return comparison.

Table 1. Frequency of Explicit Reference

		If Comparison		**Return**	
Attribute	Example Value	N	% Explicit Reference	N	% Explicit Reference
id	103	8	62.5	8	75.0
label	a43	9	22.2	8	50.0
name	apple	19	21.1	8	37.5
calories	95	7	57.1	16	56.3
color	red	9	77.8	8	62.5
shape	round	8	75.0	8	50.0
status	fresh	7	57.1	10	70.0
texture	smooth	6	66.7	8	75.0

Fisher's exact test was applied to determine if the relative proportions of the attribute (e.g. *label*, *color*, *texture*) are independent of whether they are explicitly referenced in the required expression.

The analysis revealed significance of dependence for expressions appearing in the if statement comparison (p = 0.028) but not for those appearing in the return statement (p = 0.769).

We also employed Fisher's exact test to verify if the use of the *style* attribute occurred for tasks calling for particular attributes. The analysis revealed a significance of dependence (p = 0.003) across both the comparison and the return statement.

4.4 Discussion of Initial Analysis

We speculated that the student participants would consider the following attributes (with example values) as identifying: *id* (103), *label* (a43) and *name* (apple). Consistent with the use of metonymy, we anticipated that students would not explicitly reference these identifying attributes (e.g. referencing **obj** instead of **obj.name**) when constructing expressions in their programming code.

With the exception of the *id* attribute, the student reference constructions were consistent with our metonymy-based predictions. Both the identifying name attribute and label attribute were explicitly referenced less frequently than the putatively descriptive attributes. This trend was present in both the **comparison** expressions and the **return** expressions, although we only found a statistically reliable dependence with the comparison expressions.

Our search for an explanation of the exceptional effect of the *id* attribute led us to the instructions given the students. Of the experimentally-controlled attributes, only the *id* attribute was used as an example in the task instructions. Our conjecture is that students drew upon this example when constructing expressions referencing the object's *id*. This interpretation is consistent with the appearance of *style* attribute in many of their expressions. Even though none of the experimental tasks asked students to construct expressions using the style attribute, it nonetheless appeared in many of the expressions. As we noted, it appeared most frequently in the tasks that called for a reference to the *texture* attribute, presumably because its example value (smooth) could be considered as a description of the object's style. In any case, the unsolicited appearance of the style attribute in the coded expressions provides evidence that students may mechanically use expressions from an example in the construction of their coded solution. For tasks calling for the construction of an expression with the texture attribute, the example leads to an incorrect solution (albeit with the attribute explicitly referenced). For tasks calling for the *id* attribute, the example leads to a solution that is correct with respect to both the explicit reference and the attribute itself.

4.5 Additional Analysis

To further test whether our hypothesized attribute categories lead to explicit references, we merged the attributes into three categories: identifying (i.e. name and label), descriptive (i.e. calories, color, shape, status and texture) and example-supported (i.e. id). We then constructed a mixed effects generalized linear model that included attribute category and statement type (i.e. comparison vs. return statement) as fixed effects. The participant was included as a random effect. To account for categorical prediction (explicit reference vs. implicit), the linear model used a logit function and assumed a binary distribution. The attribute category yielded a significant fixed effect, F(2, 67) = 5.82, p = 0.0047 while the statement type did not yield a significant effect, F(1, 67) = 0.38, p = 0.540. Finally, we performed pairwise tests for the attribute categories. Table 2 shows resulting t statistics for each of the three comparisons, revealing that both example-supported attributes and descriptive attributes have a significant effect over identifying attributes when it comes to producing an explicit reference in the coded solutions.[3]

[3]We report unadjusted p values. We note that our analysis yields significant differences even when adjusted using a post-hoc Bonferroni treatment.

Table 2. Comparative Effects of Attribute Type

Attribute Category	t (DF = 67)	p
Identifying vs. Descriptive	3.20	.0021
Descriptive vs. Example-supported	0.42	.6763
Identifying vs. Example-supported	2.43	.0178

5 GENERAL DISCUSSION

The goals of our study were to 1) identify when reference-point errors occur, 2) offer instructional strategies for addressing them and 3) provide additional insights on theoretical sources for these errors. In this section we revisit these goals and address them.

5.1 When reference-point errors occur

Our results show that students more frequently specify the attribute if it is used in an instructional example. For tasks calling for the *id* attribute, which was used in an example, they more frequently included the attribute in the reference. Even when the task did not call for the *style* attribute, also used in an example, it frequently appeared as an attribute in the solution, particularly for tasks involving attributes (e.g. *texture*) semantically related to style. This interpretation is consistent with prior research that theorized how students draw upon code examples to produce the solution [11, 25, 29]. For example, Pirolli and Recker [25] found that their subjects produced fewer errors when working on novel tasks that were supported by examples than on tasks that were not. For our study, the supporting examples made it more likely that the student provided the explicit attribute, even if it was not the attribute required for the task.

For cases without a supporting example, our results produced errors consistent with the use of metonymy [18, 19], previous empirical results [17, 18], and less formal observations of student errors [9, 12, 15]. So far, studies have revealed a prevalence of these reference-point errors for novice programming. Consistent with previous studies, the students in our study were novices, reporting a median of two prior programming courses. To our knowledge, there is no study on the occurrence of reference-point errors for skilled programmers. The prevalence of these errors may incrementally decrease with practice or there may be an abrupt transition when reference-point errors all but disappear.

5.2 Learning Implications and Instructional Strategies

One concern with the use of examples is that they may not necessarily support learning. Students may substitute segments of example code to produce the solution without any reflection. In these scenarios, students are constructing a "literal" reference, not by a deliberate strategy, but by a rote copy and paste of a working solution. Since this form of mechanistic construction is done without attending to relevant relations for transferable learning, this form of problem-solving may not yield useful schema acquisition. This analysis is consistent with cognitive load theory, where the overhead of problem-solving detracts from learning [35]. On the other hand, students may choose to engage in learning behaviors that are not intrinsic to solving the problem or, in this case, constructing a correct reference. Such behaviors include reflecting on the example or self-explaining how the example works in the problem-solving context. Theoretical work and empirical studies have shown how these behaviors promote learning [5]. Moreover, Pirolli and Recker report that verbal protocol analysis of study participants reveals fewer errors among participants with more events of self-explanation and reflection. Additional research on self-explanation shows that coaching students to self-explain produces increased learning [6]. With respect to forming correct references, instructors may want to coach students to explain their reference constructions. A possible device for eliciting explanations may involve teaching students about metonymy and asking them to use it as a structure for explaining how they need to construct a "literal" reference required for correct code.

As an alternative to eliciting explanations while coding, instructor-led explanations of worked examples may provide better support to learning, particularly if done with explicit reference to all attributes. Guided instruction may also include explaining how metonymy operates in human communication yet yields incorrect references for the computer. Such instruction could be integrated with live coding, which has been shown to facilitate student learning [30]. Other forms of guided instruction such as subgoal labeling [22] may also be effective here. For reference construction, it could include explicit checks on whether the literal construction is present.

Finally, while not emphasized in this paper, there may be some benefit to asking students to compare their answer with a correct solution. As we reported elsewhere [20], some students noticed their mistake when presented a correct solution and then indicated that they forgot to explicitly mention the attribute (e.g. "I also forgot the .label"). This reflective component may prompt the necessary knowledge acquisition and conscious avoidance of the pitfalls of metonymy necessary for successfully instantiating the full reference in the future.

5.3 Sources of difficulty

In the introduction, we surveyed multiple knowledge sources and systems that might account for the reference-point errors found in student-constructed references. While our results do not conclusively rule out any of these sources, we offer some interpretations that account for some of the differences among the conditions.

One possibility is that students conceptually misrepresent the object, conflating its identifying attribute with the object itself. This interpretation immediately explains why the identifying attribute is omitted when constructing a reference. We can also understand how the identifying attribute then appears in the reference when it draws upon a supporting example: the student may just be copying relevant code into the solution. Less clear is whether this interpretation explains why the identifying attribute might appear more frequently in some contexts but not others. Whether the context concerns complexity or the coding construct (e.g. conditional statement vs. return statement), drawing upon an impoverished mental representation should produce incorrect references with equal frequency.

We have also considered the possibility that students have a misconception of what the notional machine can do. They may believe that the system can resolve reference-point shifts, just like humans. While we have not seen any evidence of this understanding in student explanations, it does explain why students would omit the identifying attribute, thinking that the system will be able to infer that they intend to refer to the attribute and not the object itself. This interpretation is also consistent with the observation that identifying attributes appear in references that have supporting examples. Rather than require the system to do the work, they may just find it easier to copy in some working code. Most problematic for this interpretation is that reference-point errors may appear more frequently in some coding contexts. While it is possible to construct models of a notational machine that work in one particular way for some coding contexts and a different way for other contexts, it requires a less parsimonious theory.

The third possibility for explaining the observed reference-point errors is that students are relying on well-practiced habits of communication. Drawing on this knowledge explains why identifying attributes are omitted, a practice consistent with metonymy. For tasks that draw on examples, students may simply use the example code and thus circumvent application of the habits of communication—an explanation of why identifying attributes sometimes appear in the code. Finally, it offers an explanation why attributes may be omitted under different coding contexts. One possibility, as an effect of cognitive load, is that the complexity of the expression inhibits application of the less efficient deliberate knowledge system and, in its absence, the more efficient practiced knowledge applies. Another possibility is that the different semantics of the conditional statement and the return statement elicits differing likelihoods that knowledge in the efficient system applies. In any case, this interpretation suggests that reference-point errors are not misconceptions, that is, mistaken beliefs about the task or notional machine, but procedural habits that novice programmers have practiced throughout their lives. Just as a Stroop error is not a problem of understanding the task (subjects know that they need to identify the color) or missing competencies (subjects know how to recognize colors and say their name), constructing a literal reference may not involve any misconceptions. Instead, it involves skill-based slips that are the result of years of practice. With this interpretation, "learning to get literal" may not require correcting misconceptions but rather introducing strategies for noticing errors and allowing enough practice so that effective, literal reference construction happens with minimal effort.[4]

At this time, the evidence for a prevalence of reference errors in complex expressions is inconclusive. While we observed more errors in the more complex condition, the difference was not statistically reliable. It is also possible that it is not the complexity of the statement but rather the type of statement that inhibits whether practiced habits apply. For example, a student may interpret the *if* statement as having a certain selectional semantics for which just referencing the object is appropriate. On the other hand, other work [18] also found a higher incidence of errors in the more complex expressions even though the difference was not systematically explored and analyzed. Stronger evidence of a complexity effect could be obtained by conducting studies that systematically vary statement types and expression complexity to see where reference-point errors are most likely to occur.

We may also want to look at student comments for insight on whether they are working with a misconception or simply slipping on habits of communication. For example, some students may report their mistake as a lapse of thinking (e.g. "I forgot the .label") as opposed to a misunderstanding of the object (e.g. "I didn't realize that the object has its own 'title' attribute") or what the notional machine can do (e.g. "I thought the program would understand that I was really referring to the name"). While we do not expect students to have access to their practiced, proceduralized knowledge, their explanations may provide some indications (e.g. 'forgot' vs. 'didn't realize') as to whether they are habits of practiced routines or explicit misunderstandings of the task or system.

Our long-term goal is to further explore factors that yield reference errors in the context of a predictive process-oriented model. At this point, we conjecture that its operational knowledge incorporates the following:

- Procedures that draw upon example segments.
- Procedures for correctly constructing the attribute reference but are more likely to be inaccessible for more complex expressions.
- Default communication procedures that use whole objects to represent identifying attributes.

Future work calls for its development as a working computational model, possibly representing the above knowledge as productions in the context of a cognitive architecture such as ACT-R [1] or

[4]Interestingly, Besner, Stolz and Boutilier [3] report a variant of the Stroop task where human subjects are able to improve their relative performance on incongruent tasks.

Soar [23]. The development of the model may draw upon a previous modeling effort which applied verbal reasoning knowledge for solving syllogisms [26].

6 CONCLUSION

In this paper we explored factors that yield reference-point errors in novice programming. Drawing upon the study presented in this paper and previous analysis [18, 20], we hypothesize that reference-point errors occur more frequently for the following conditions:

- The relevant attribute is an identifying attribute (consistent with the use of metonymy).
- The relevant attribute lacks a supporting example.
- The relevant attribute is used in a relatively complex expression.

The present study revealed statistically reliable evidence that supports the first two factors. With respect to the third factor, we seek further confirmation that an expression's complexity yields more reference errors.

Theoretically, we have considered knowledge sources that underlie the observed reference-point difficulties. Our experimental results do not allow us to rule out any of the candidate sources. In fact, a novice programmer may draw upon any one of them given that person's prior practice, instruction, and the context of the task. However, we suggest that a novice programmer's reliance on well-practiced habits of communication may account for a large proportion, if not the majority, of such difficulties. This theory accounts for all observed behavior and is consistent with established theories of human cognition.

Practically, we have have offered instructional strategies for addressing reference-point errors. Regardless of the theoretical source of the difficulty, these strategies should be effective in that they all support the student noticing the errors and offer opportunities to practice working on them. These strategies are also well supported in prior research for diverse difficulties.

An open question is whether a model of reference construction in computer code will employ the same mechanisms as those that underlie metonymy in human-to-human communication. Another question is whether we can produce an operational method for determining whether an attribute plays an identifying role. A study of such methods may be useful for investigating the extent to which the construction of reference-point errors in computer programming parallels the use of metonymy in human communication. A final direction for future exploration is whether reference-point errors occur in other areas of human-to-computer communication that are not code-based. For many user interfaces, a user must specify the object of an action. If the point of reference (e.g. tree or branch) is ambiguous given the action (e.g. change color), we might see reference errors that are similar to what we observed with novice Python programming.

REFERENCES

[1] John R Anderson. 1993. *Rules of the mind.* Lawrence Erlbaum Associates, New York.

[2] Piraye Bayman and Richard E. Mayer. 1983. A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements. *Commun. ACM* 26, 9 (Sept. 1983), 677–679.

[3] Derek Besner, Jennifer A Stolz, and Clay Boutilier. 1997. The Stroop effect and the myth of automaticity. *Psychonomic bulletin & review* 4, 2 (1997), 221–225.

[4] Jeffrey Bonar and Elliot Soloway. 1985. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human–Computer Interaction* 1, 2 (1985), 133–161.

[5] Michelene TH Chi, Miriam Bassok, Matthew W Lewis, Peter Reimann, and Robert Glaser. 1989. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive science* 13, 2 (1989), 145–182.

[6] Michelene TH Chi, Nicholas De Leeuw, Mei-Hung Chiu, and Christian LaVancher. 1994. Eliciting self-explanations improves understanding. *Cognitive science* 18, 3 (1994), 439–477.

[7] M. Clancy. 2004. Misconceptions and attitudes that interfere with learning to program. In *Computer science education research*, Sally Fincher and Marian Petre (Eds.). Taylor and Francis Group, London, 85–100.

[8] William Croft. 1993. The role of domains in the interpretation of metaphors and metonymies. *Cognitive Linguistics* 4, 4 (1993), 335–370.

[9] Janet Davis and Samuel A. Rebelsky. 2007. Food-first computer science: starting the first course right with PB&J. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*. ACM, New York, NY, USA, 372–376.

[10] Benedict Du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.

[11] Kathi Fisler and Francisco Enrique Vicente Castro. 2017. Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 12–20. DOI: http://dx.doi.org/10.1145/3105726.3106183

[12] Simon Holland, Robert Griffiths, and Mark Woodman. 1997. Avoiding object misconceptions. *SIGCSE Bull.* 29, 1 (1997), 131–134.

[13] Daniel Kahneman. 2011. *Thinking, Fast and Slow*. Farrar, Straus and Giroux, New York.

[14] George Lakoff and Mark Johnson. 1980. *Metaphors We Live By*. The University of Chicago Press, Chicago, IL.

[15] Gary Lewandowski and Amy Morehead. 1998. Computer science through the eyes of dead monkeys: learning styles and interaction in CS I. *SIGCSE Bull.* 30, 1 (1998), 312–316.

[16] Colin M MacLeod. 1991. Half a century of research on the Stroop effect: an integrative review. *Psychological bulletin* 109, 2 (1991), 163–203.

[17] Craig S. Miller. 2012. Metonymic errors in a web development course. In *Proceedings of the 13th annual conference on Information technology education (SIGITE '12)*. ACM, New York, NY, USA, 65–70.

[18] Craig S. Miller. 2014. Metonymy and reference-point errors in novice programming. *Computer Science Education* 24, 3 (2014).

[19] Craig S Miller. 2016. Human language and its role in reference-point errors. In *27th Annual Workshop of the Psychology of Programming Interest Group - PPIG 2016*.

[20] Craig S. Miller and Amber Settle. 2016. Some Trouble with Transparency: An Analysis of Student Errors with Object-oriented Python. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. ACM, New York, NY, USA, 133–141. DOI: http://dx.doi.org/10.1145/2960310.2960327

[21] Lance A Miller. 1981. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal* 20, 2 (1981), 184–215.

[22] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. 2015. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15)*. ACM, New York, NY, USA, 21–29. DOI: http://dx.doi.org/10.1145/2787622.2787733

[23] Allen Newell. 1994. *Unified theories of cognition*. Harvard University Press.

[24] Klaus-Uwe Panther and Günter Radden. 1999. Introduction. In *Metonymy in language and thought*, Klaus-Uwe Panther and Günter Radden (Eds.). Amsterdam: John Benjamins, 1–14.

[25] Peter Pirolli and Margaret Recker. 1994. Learning strategies and transfer in the domain of programming. *Cognition and instruction* 12, 3 (1994), 235–275.

[26] Thad A Polk and Allen Newell. 1995. Deduction as verbal reasoning. *Psychological Review* 102, 3 (1995), 533.

[27] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: a literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1.

[28] Bárbara Eizaga Rebollar. 2015. A Relevance-theoretic Perspective on Metonymy. *Procedia-Social and Behavioral Sciences* 173 (2015), 191–198.

[29] Robert S Rist. 1991. Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. *Human-Computer Interaction* 6, 1 (1991), 1–46.

[30] Marc J Rubin. 2013. The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 651–656.

[31] Juha Sorva. 2008. The same but different students' understandings of primitive and object variables. In *Proceedings of the 8th International Conference on Computing Education Research*. ACM, 5–15.

[32] Juha Sorva. 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education (TOCE)* 13, 2 (2013), 8.

[33] James C Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM* 29, 7 (1986), 624–632.

[34] J Ridley Stroop. 1935. Studies of interference in serial verbal reactions. *Journal of experimental psychology* 18, 6 (1935), 643.

[35] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

Table 3. Week-by-week outline for the first-quarter Python course

Week	Topics
1	Intro to CS, expressions, basic data types
2	Basic Input and output, Control flow: If statement and simple for loops, calling and defining functions
3	Lists and strings, type conversions, modules
4	File input/output: Writing, reading, and appending to files, errors and debugging
5	Review and midterm
6	Exceptions: catching exceptions, raising exceptions, and using exceptions
7-8	Programming patterns: accumulator loops, indexed loops, nested loops, multidimensional loops, while loops
9	More types: Dictionaries, sets, tuples
10	Namespaces: Program stack and scope: local/global variables

[36] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming Misconceptions for School Students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research.* ACM, 151–159.

[37] Josh Tenenberg and Yifat Ben-David Kolikant. 2014. Computer Programs, Dialogicality, and Intentionality. In *Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14).* ACM, New York, NY, USA, 99–106. DOI: http://dx.doi.org/10.1145/2632320.2632351

7 APPENDIX

7.1 Courses

Students were recruited from two courses required in programming-intensive undergraduate degrees at our institution. It should be noted that our institution has 11-week quarters, with ten weeks for instruction and one week for final exams. The first course is the second in a sequence designed for novices. It has as a prerequisite a first-quarter Python course, which is a procedural problem-solving course. The first-quarter Python course has no prerequisites and has a week-by-week outline as given in Table 3.

The main goal of the second-quarter Python course for novices is to teach object-oriented programming. Several applications of object-oriented programming are shown in addition to recursion. The week-by-week outline of the second-quarter Python course is provided in Table 4.

An accelerated Python course is the second course from which students were recruited for this study. The accelerated Python course covers the majority of topics in the first- and second-quarter novice Python courses but condensed into a single 11-week quarter. Students are required to have completed one course in a high-level programming language which allows for the accelerated pace of the course. The week-by-week outline for the accelerated Python course is provided in Table 5.

The object-oriented programming topics in the second-quarter Python course and the accelerated Python course discuss defining classes, including attributes and methods, as well as inheritance and its implications for overloaded methods. In the second-quarter Python course the definition of operators for classes are covered, although there is not sufficient time in the accelerated Python

Table 4. Week-by-week outline for the second-quarter Python course

Week	Topics
1	Namespaces and scope; an introduction to object-oriented programming
2	Object-oriented programming, including inheritance
3	Object-oriented programming
4	Graphical-user interfaces
5	Recursion and the midterm
6	Recursion, sorting, and searching
7	Recursion
8	An introduction to HTML and WWW application development
9	WWW application development
10	WWW application development and optional topics (databases, evaluation of function efficiency, etc.)

Table 5. Week-by-week outline for the accelerated Python course

Week	Topics
1	Basic data types (including lists, strings), basic input, defining functions, basic for loops, basic control structures
2	File input/output, formatted output, modules, introduction to programming patterns: iterated and indexed loops
3	More programming patterns: accumulator loops, nested loops, multidimensional loops, while loops
4	More programming patterns: infinite loops, and exceptions
5	Modules, more types: dictionaries, and the midterm
6	More types: sets, tuples, and recursion
7	Recursion
8	Recursion and object-oriented programming
9	An introduction to HTML and WWW application development
10	WWW application development

course to do the same thing. In the second-quarter Python course multiple applications of object-oriented programming, including both GUIs and web crawlers are discussed. In the accelerated Python course there is only time to discuss web crawlers.

If advised correctly, students in the second-quarter Python course and the accelerated Python class should have no more than one previous programming course at the time of the study. But some of these students have experience from secondary school, so the number provided in the responses to the question on the study about previous programming courses was higher than might be expected.

Students were recruited for the study near the end of the quarter, so that both the second-quarter Python students and the accelerated Python students would have seen how to write classes in Python prior to the study.

7.2 Code

Below is the code for the Item and Catalog classes. The students were asked to write a lookup method for the Catalog class.

```
class Item:
    'an Item class for the catalog'
    internal_id = 100
    def __init__(self):
        Item.internal_id += 1
        self.id = Item.internal_id
        self.quantity = 0

class Catalog:
    'a simple Catalog class'
    def __init__(self):
        'the constructor'
        self.items = []

    def add(self, item):
        'add an item to the catalog'
        return self.items.append(item)

    def has_style(self, desired_style):
        '''returns true if catalog has an
              item of the given style'''
        for obj in self.items:
            if obj.style == desired_style:
                return True
        return False

    def size(self):
        '''return the number of items
             in the catalog'''
        return len(self.items)
```