3-3-2015

Learning Object-Oriented Programming in Python: Towards an Inventory of Difficulties and Testing Pitfalls

Craig Miller
DePaul University, cmiller@cs.depaul.edu

Amber Settle
DePaul University, asettle@cdm.depaul.edu

John Lalor
DePaul University, jlalor@mail.depaul.edu

Learning Object-Oriented Programming in Python: Towards an Inventory of Difficulties and Testing Pitfalls

Craig S. Miller Amber Settle John Lalor
School of Computing
DePaul University

March 3, 2015

Abstract

We report a small yet detailed study where we recorded students completing an object-oriented programming exercise in the context of a CS2 course using Python. All students struggled while completing the assignment, most notably experiencing difficulties with parameters and referencing elements with object-dot notation. While previous research has identified these areas as troublesome for novice programmers, our analysis suggests that parameters and reference specifications are particularly critical prerequisites for learning advanced object-oriented concepts with the Python programming language. Given our findings, we recommend extensive practice with parameter passing and object-dot notation before addressing advanced object-oriented concepts in a Python course.

Introduction

Increasingly the Python language with its interactive environment is being used in introductory programming courses at the university level. One source from the computing education community estimates its rise of use in CS1 courses at 40% per year (Dierbach, 2014). The appeal of Python is in its relatively simple syntax (Dierbach, 2014; Goldwasser & Letscher, 2008; Agarwal & Agarwal, 2005), which allows for more focus on problem-solving in early courses (Newhall et al., 2014). There is evidence that teaching Python in the first course is at least as good as teaching any other high-level programming language. In particular, one study found that students who had a first course in Python and then went on to a course in C++ did just as well as students who had taken C++ as their first course (Enbody & Punch, 2010). Some authors have advocated using Python in introductory object-oriented courses, mentioning its built-in classes, consistent object model, and support for inheritance as pluses of using the language (Goldwasser & Letscher, 2008). Interestingly a recent meta-review found that pass rates for courses taught in any particular

Contact email address: cmiller@cdm.depaul.edu

language are not significantly different (Watson & Li, 2014), which suggests that the language used may not make as much difference as educators believe when it comes to student success. There is evidence, however, that the use of Python in conjunction with mentoring programs and other retention approaches can be helpful in improving the diversity of students in computer science programs (Newhall et al., 2014). Recently researchers have been particularly interested in developing web-based environments for Python programming since these remove any software-related barriers for novice programmers (Edwards, Tilden, & Allevato, 2014).

While Python has grown in popularity, an understanding of object-oriented programming remains important for computer science students. The process of learning to write object-oriented programs is simultaneously important and difficult enough that researchers have devised systematic programming techniques for novices learning the paradigm (Caspersen & Kölling, 2006). There have been many reports on students having difficulty learning object-oriented principles including inheritance (Liberman, Beeri, & Ben-David Kolikant, 2011). In addition to difficulties understanding the inheritance model, students also have difficulty successfully referencing the needed item, often indicating the attribute when the whole object is needed and vice versa (Holland, Griffiths, & Woodman, 1997; Miller, 2014). A study of student visualizations of object-oriented programs revealed that students commonly misunderstand the relationship between methods and objects and how methods operate on attributes, and misidentify objects with attributes (Sajaniemi, Kuittinen, & Tikansalo, 2008). One pair of researchers summarized existing knowledge of students problems with object-oriented programming and compared the results against student programs, confirming errors in basic mechanics, problems with linking and interacting, instance/class conflation, class/collection conflation, problems with hierarchies and abstraction, identity/attribute confusion, problems with encapsulation, and failures in modeling as the most frequent problems, but failing to find issues with accessors/mututators or constructors, class/variable conflation, or identity/attribute conflation (Sanders & Thomas, 2007).

While there is significant debate surrounding the teaching (and learning) of object-oriented programming (Berglund & Lister, 2010; Lister et al., 2006), it is clear that for some educators the goal remains to introduce students to the paradigm early. For institutions where Python is the introductory language, this means that the object-oriented features of the language are relevant. To summarize, the Python features most relevant for learning object-oriented principles include the following:

- An interactive environment that allows students to easily load code and test individual statements (Goldwasser & Letscher, 2008). Consequently, low overhead allows for more testing with each change, which encourages experimentation. It also allows instructors to avoid using external tools early (Goldwasser & Letscher, 2008).

- Python supports learning the imperative paradigm first and permits user-defined classes and inheritance to be taught later. At the same time, a consistent object model makes OO programming seem natural (Goldwasser & Letscher, 2008).

- Python provides a simpler and arguably more transparent execution model for OO and inheritance. Key features are:

- No significant difference between functions and methods.
- The object of a method is passed as the first parameter. Methods are explicitly defined with the first parameter being the object, called self by convention. This draws attention to the distinction between instance scope versus local scope(Goldwasser & Letscher, 2008).
- As with all variables, instance variables are not defined until they are assigned, which can only be done from inside methods. All instance data is explicitly referenced from the self object.
- Python supports single and multiple inheritance with minimal syntactic overhead(Goldwasser & Letscher, 2008).

This situation suggests the principal research question driving this work: How does this context affect student learning of OO concepts? Our goal for exploring the question is to develop an inventory of likely student misunderstandings and an understanding of how students attempt to resolve them using Python's interactive environment. In the process we apply additional analyses to identify critical prerequisites for the successful learning of advanced OO concepts with Python.

Exploratory Study

In this section we present an exploratory study of student learning in an interactive Python environment. We recorded five students working on an exercise using inheritance. In addition to capturing all screen activity, we followed a think-aloud protocol as presented in Ericsson and Simon (1993). Using instructions such as "say whatever words come to your mind," the protocol asks participants to think aloud without asking them to explain or reflect on their actions. Ericsson and Simon present findings showing that such protocols provide insight into participant actions without affecting behavior at the cognitive level.

Design and Context

For observing student difficulties involving inheritance, we asked students to convert a stack class from being a subclass of object to a subclass of list in Python. While this exercise arguably has pedagogical drawbacks (e.g. inheriting methods from the list class has a dubious benefit), the exercise is relatively self-contained, adequate for eliciting difficulties, and common in introductory courses. We note that the course text (Perkovic, 2011) presents a queue class that is developed as a subclass of object. As a class exercise, many course instructors then develop the queue class derived from the Python list class and make comparisons between the two versions.

Students were recruited from CSC 242: Introduction to Computer Science II, which is a second-quarter programming course designed for novices who are majoring in a development-focused program. During the time of the study this included computer science, computer game development, and information assurance and security engineering majors. Students who have programming experience take a separate, accelerated development course in Python which condenses much of the material in the two classes into a single quarter. Regardless of which Python sequence they take, students go on to either learn Java or C++ depending on their major.

The institution in which the participants are enrolled has 11-week terms. The material covered in CSC 241: Introduction to Computer Science I, which is a prerequisite for CSC 242, focuses on problem solving, algorithm development, and structured programming. The students learn basic types, decision structures, looping structures, string processing, the use of and definition of Python functions, collections, and specialized modules such as math and random. Although students in CSC 241 use built-in objects and their methods, they do not define any classes of their own. The second-quarter Python course focuses on object-oriented programming, applications of object-oriented programming such as the creation of GUIs and the development of a web crawler, and recursion. The first three weeks of the quarter are spent discussing object-oriented programming isolated from more complex applications, including how to define classes and inheritance. The stack class conversion activity was timed so that students would be completing it immediately after completing the portion of the class discussing inheritance. In some cases, this means that they were completing the activity prior to having had an assignment on inheritance. The activity was also typically completed prior to the midterm exam, which takes place during the fifth or sixth week of the quarter.

Given a small sample, our goal is not to draw inferences such as quantitative estimates of student difficulties. Instead, the approach is similar to a formative usability study, where the goal is to create an inventory of common issues in comparable contexts of learning. With this goal in mind, a sample of five protocol cases is adequate for observing common problems in at least one of the sessions (Sauro & Lewis, 2012). This approach also permits refinements to the protocol and task in order to elicit a broader range of issues. As we will show, insight is not drawn from quantitative analysis nor by controlling presentations of the problem, but from detailed observations of what the students do and say. When combined with findings from other studies, the details of the student actions and the verbal protocol indicate *why* students encounter certain difficulties and permit further analysis for understanding how features of the Python language and development environment affect student learning.

Method

Members of the research team went into the second-quarter Python classes, and students were given a brief explanation of the project. They were then offered the chance to fill out an interest form. All students were given forms and all forms were collected, so that there was no pressure on students to participate. Each student who indicated interest was emailed to schedule time to take part in the study. Of the students that indicated interest, five students responded to the scheduling requests to participate.

Each of the five participants was read an introductory script prior to beginning the exercise. The script included an overview of the study, the intended goal, and what should be done to complete the exercise. The study problem was the stack conversion exercise explained in the previous section.

The students were given time to re-read this script and ask any questions prior to beginning the study. Once they finished reading the script, the recording was started so that the student's consent could be recorded. Audio and screen activity were recorded during the session. As the students worked through the problem, they were instructed to talk out loud while working through the problem, so that the steps they were taking as

Table 1
Participants

Pseudonym	Age	Courses	Week
Alex	18	1	9
Ben	20	2 or 3	5
David	19	2	6
Jay	20	1	4
Tom	25	2	5

they worked through the problem could be recorded. If the student stopped talking aloud as they worked, they were encouraged to say what they were thinking or asked to speak up.

There were two slight variations of the study protocol. With the first three participants, given the pseudonyms Alex, Ben, and David, the version of the problem presented included a completely implemented stack class as a subclass of object. Participants were instructed to modify the stack class to be a subclass of list and then to indicate whether each method should be deleted because it was redundant, modified because it was incorrect, or left alone because it was correct in the new implementation. The researchers simply allowed the students to work uninterrupted on the activities even if the participants became confused or lost about what to do next.

During the second phase of the study, which includes participants with pseudonyms Jay and Tom, the protocol was modified slightly. Participants were given a completely implemented stack class as a subclass of object. They were also given a version of the stack class implemented as a subclass of list in which each method body only contained a placeholder pass statement. The participants were instructed to complete the stack class written as a subclass of list by providing bodies for each of the methods and were instructed to not add or delete any of the methods. Further, questions that the students had were answered by the study facilitator. If the students were having difficulty with the problem, hints were provided after set time intervals to help the student with ways to work through the problem.

In both versions of the protocol, at the end of the allocated time (around 30 minutes) students were asked follow-up questions about the problem, any difficulties that they had, and what they thought the relevance of the study was. Finally, students were asked their age, gender, and the number of programming classes that they had taken. Each student who participated was given a $25 Amazon gift card for participating in the study. Table 1 gives the details for each participant in the study, and also notes which week of the 11-week quarter the participant took part in the study.

Results

The five protocol cases were reviewed by each of the authors. Notes of events were shared and then iteratively integrated to produce a detailed log of event types, typed text, verbal comments and interpretations. Table 2 shows a sample of logged events for one of the sessions. A total of 215 events were logged, mostly consisting of edit events (N=71) and test events (N=79).

Table 2
Sample event log for Jay

Minute	Event type	Target	Typed text	Note
6	edit	size method	return len(self.s)	
6	edit	isEmpty method	return self.size()==0	
7	test	constructor	st=stack()	
7	test	isEmpty method	st.isEmpty()	produces an error since isEmpty calls size and size refers to self.s
7	read	instructions		
8	quote			sees error *'stack' object has no attribute 's'*
8	edit	pop method	return self.s.pop(0)	
8	edit	push method	return self.s.insert(0,item)	

Difficulties

Drawing upon the recordings and event logs, we identified major difficulties that the students experienced while working on the inheritance exercise. Below we enumerate these difficulties and indicate by pseudonym which students encountered them.

1. **Unintentionally overriding the constructor.** Indicates that an overridden method is called first and then the superclass method. (Alex)

2. **Misaligning parameters.** Misaligns or omits the self object in the parameter list. (Alex, Jay)

3. **Specifying incorrect reference.** Reference-point misunderstanding, e.g. self vs. self.s. (Alex, Jay, Tom; possibly Ben and David, but they never tried to change the references in the function)

4. **Calling method without parentheses.** Tries to call a method without the parentheses around parameters, for Python 3.x. (Ben)

5. **Using two return statements to return two sets of values.** Writes a method with two return statements to produce both values. (Ben)

6. **Conflating output from a return statement and output from a print statement.** Writes a method with a print statement to return a requested value when the assignment calls for a return statement. (David)

7. **Failure to test after significant edits.** Makes significant edits to method(s) but fails to test the edits before writing more code, resulting in incomplete verification of the changes. (David, Jay)

Detailed Accounts

Below are selected, detailed accounts exemplifying key difficulties from the previous section.

Alex overrides the constructor. After Alex defines the header for the new class, he indicates that the `init` method (Python constructor) is not needed, saying, "I don't need the init since it is already init-ed with the list." Yet, he defines the `init` method in the new derived class adding a `pass` statement (Difficulty 1).

Alex then loads the code module and tests the constructor, specifying initial contents for the stack: `myStack([1,2,3])`. The test produces an error indicating that there is a mismatch in the number of parameters since the newly created object is passed as the first parameter, followed by any additional parameters (Difficulty 2). Alex eventually makes progress by seeing an example that invokes the constructor without any additional parameters. This invocation produces a test without any errors, although Alex fails to provide an explanation as to why no additional parameters should be used.

Alex questions whether `pass` should be used in the constructor. He reviews the `init` method in the `list` documentation and sees 'x' used in the documentation:

```
x.__init__(...x) initializes x
```

Alex concludes that his constructor needs to initialize 'x' but he mistakenly confuses x as the object and x as an attribute of the object (Python effectively implements instance variables as attributes of the self object):

```
self.x = []
```

By conflating `self` and `self.x` (Difficulty 3), Alex then defines push and pop in terms of `self.x`. These definitions produce working code, but they do not make use of any methods inherited from the list class. Alex rereads the task instructions and considers that the `init` method may be unnecessary, but deleting it produces errors since his push and pop methods are defined in terms of `self.x`. Consequently, he restores his definition of `init`. In the end, there is no verbal recognition or explicit resolution of the three misunderstandings that Alex encountered.

Jay grapples with using self. Jay begins his second implementation of the derived class of list. He writes

```
self() == 0
```

for the body of the isEmpty method (Difficulty 3). He verbalizes his uncertainty about his solution. After completing the class he begins testing and is unable to debug the isEmpty method.

Five minutes later Jay asks for help. In his interaction with the researcher he correctly verbalizes that self is a list object. He changes the body of isEmpty to:

```
self == 0
```

His tests reveal errors with pop, and he changes the method body to:

```
self.remove(self)
```

(Difficulty 2). Further editing results in the body of isEmpty becoming

```
self == 0
```

After further testing reveals errors, Jay modifies the isEmpty method to be

```
return self(item) == 0
```

(Difficulty 3).

A further hint comes from the researcher at minute 26. In quick succession Jay changes the body of isEmpty to:

```
return self(size) == 0
```

and then to:

```
return size() == 0
```

which represent two instances of Difficulty 3. After more testing he writes the correct version of the isEmpty method, which is

```
return self.size() == 0
```

Ben tries to call a method without parentheses. Ben attempts to test a special member function with the following syntax:

```
st.__str__
```

The Python console produces information *about* the method (parentheses are required to call the method). He similarly 'tests' `__repr__` and `__iter__`, but there is no indication that he understands that he is not actually calling these methods (Difficulty 4). Ben concludes that the `repr` and `iter` methods are "redundant" and comments them out from the code. Ten minutes later, he returns to the `str` method and tries to use it (again) without the parentheses. Ostensibly not satisfied with the results, he modifies the code in the method definition multiple times, each time 'testing' without the parentheses. The exercise ends without Ben realizing that he was not calling the method for his testing.

Discussion

The exercise for revising the stack class was expressly designed to teach inheritance concepts, yet we observed little time spent addressing inheritance. Only Alex was able to consider inheritance concepts. Even then, it is not clear whether he finally understood that defining the constructor in his stack class overrode the needed functionality in the base class.

In place of inheritance concepts, we observed that all students struggled with more fundamental concepts and misunderstandings. Most notable were difficulties with parameter alignment and referencing the correct element with object-dot notation. In the next sections we discuss these difficulties further. While these two areas of difficulty are well documented as troublesome for students, we further explore their consequences in the Python context. For our analysis, we draw upon the theoretical framework of threshold concepts (Meyer & Land, 2003), which are characterized as transformative, integrative concepts that then serve as a necessary gateway to more advanced concepts. Our analysis suggests that competence in parameter alignment and object-dot notation is more critical when learning programming with Python than with other common introductory OO languages such as Java.

Parameter passing

We observed two students showing difficulty aligning parameters when defining instance methods (Alex and Jay). Previous studies have noted that students have difficulty with parameters (Madison & Gifford, 2002; Fleury, 1991). Learning to program in Python includes all the previous issues for parameter passing, but it also introduces a new issue with respect to learning to write class definitions. Like many OO languages frequently used in CS1 courses, such as Java, an instance method may be called with the following syntax:

```
obj1.move(steps)
```

However, the method definition must be defined with an extra parameter in order to receive the instance object:

```
def move(self, distance)
```

The Python front-end processor transforms the method class so that `obj1` is passed as the first parameter. Learning to define Python classes consequently requires a solid understanding of parameter passing in order to avoid confusion with parameter alignment when class definitions are taught. Moreover, competence in parameter passing may require more than just explicit knowledge of their workings but also tacit mental structures (see Soloway & Ehrlich, 1984 for presentation and evidence of both types of knowledge in expert programmers). Students may need extensive practice with parameter passing, perhaps to the point where it is routine and effortless, before they can effectively transform the method's syntactic structure.

Dot-object notation

All students demonstrated difficulty effectively referencing the correct memory element using dot notation. Most notable was how students confused the reference to the instance object (`self`) with a reference to an attribute belonging to the instance object (e.g. `self.s`). As we have already noted, these reference-point errors are common in other languages (Holland et al., 1997; Miller, 2014; Sajaniemi et al., 2008). However, understanding class definitions with Python critically depends on competence expressing references with dot notation since Python uses these references in place of instance variables. As we observed in our protocols, difficulty specifying the correct element detracted from learning the more advanced object-oriented concepts targeted by the exercise.

Threshold concepts for Python

As we discussed, both parameters and references are critical, gateway concepts for learning how to define classes and advanced object-oriented concepts such as inheritance. Consistent with Meyer and Land's characterization of threshold concepts as being transformative and integrative, they serve as critical elements for advanced topics. For the case here, that includes instance method definitions and inheritance. Meyer and Land also describe threshold concepts as *probably* bounded, irreversible, and troublesome. Both our observations and previous studies demonstrate the troublesome nature. Yet, the required understanding for successfully working with parameters and references is arguably well bounded by the operating rules of the Python language and environment. Perhaps the

only debatable quality is irreversability, which would need to be addressed with additional study that longitudinally tracks student understanding. In any case, as gateway threshold concepts, they suggest the following:

- Evidence supporting the benefits of an imperative-first approach when using Python for introductory courses. We note that this approach is consistent with some textbooks (e.g. Perkovic, 2011).
- Need for extensive practice with parameters and reference specification, regardless of the approach (imperative-first or objects-early) taken. Increased experience with these concepts facilitates schema construction and allows cognitive resources for focusing on advanced OO constructs.

While it is common sense that material needed for later concepts should be practiced early and often, this work also suggests that the ordering of the material may depend on the language taught. For Python it may be that a spiral approach to parameter passing and dot-object notation is helpful, allowing the reintroduction of the concepts in multiple contexts.

Additional issues

We observed one student (Ben) repeatedly attempt to call a special instance function (e.g. `__str__`) without parentheses. Because Python allows reference to functions as a first-class data type, the interactive Python console does not produce an error. Instead it provides information about the function. Thinking that this information was what the function returns when it is called, Ben erroneously thought something else was wrong in the method definition. He repeatedly edited the definition to no effect. Eventually he ran out of time before realizing the problem. Despite the extensive difficulty, we do not consider function-calling notation a threshold concept. Here the resolution is fairly simple by alerting students to the possible pitfall. Nevertheless awareness of the potential difficulty could be of value for Python instructors. Instructors may consider a short demonstration showing the difference in outputs when a function is referenced with and without parentheses. Unlike the threshold concepts we just discussed, a comprehensive understanding of functions as first-class types is not necessary to make progress with the current set of topics.

Finally we observed extensive testing by all of the students. We recorded 79 testing events (e.g. typing a statement in the interactive Python console), which comprised more than a third of all logged events. Despite the facility to repeatedly test, we saw little evidence of effective testing for learning. In some cases, a misdiagnosed error can reinforce a misconception and create new ones. As we observed with Ben, a student may spend an extensive amount of time trying to fix code when the problem lies elsewhere. Less clear are effective strategies for teaching students how to make the most of a highly interactive environment. Certainly debugging strategies qualify and instructors may provide live demonstrations exemplifying their practice. Yet, effective learning involves more than just fixing broken code. It can, and probably should, include deliberate strategies involving experimentation. In any case, we see teaching effective learning strategies as an important direction for additional research, particularly in the context of interactive environments that languages like Python afford.

Conclusion

We have argued that parameter passing and dot-object notation are critical gateway concepts for effectively learning more advanced object-oriented concepts in Python. Our evidence is not just supported by observed difficulties from our study, which is limited by its small sample. It is also supported by previous studies describing difficulties in these areas and by our analysis showing how Python requires competence in parameter passing and reference specification as threshold concepts. While early introduction to OO concepts using other languages such as Java may be viable, our analysis provides evidence that an imperative-first approach when using Python may be more practical or at least suggests a need for extensive practice of parameter and reference specification before advanced OO concepts are introduced.

References

Agarwal, K. K., & Agarwal, A. (2005, April). Python for cs1, cs2 and beyond. *J. Comput. Sci. Coll.*, *20*(4), 262–270. Retrieved from http://dl.acm.org/citation.cfm?id=1047846.1047887

Berglund, A., & Lister, R. (2010). Introductory programming and the didactic triangle. In *Proceedings of the twelfth australasian conference on computing education - volume 103* (pp. 35–44). Darlinghurst, Australia, Australia: Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1862219.1862227

Caspersen, M. E., & Kölling, M. (2006). A novice's process of object-oriented programming. In *Companion to the 21st acm sigplan symposium on object-oriented programming systems, languages, and applications* (pp. 892–900). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1176617.1176741 doi: 10.1145/1176617.1176741

Dierbach, C. (2014, June). Python as a first programming language. *J. Comput. Sci. Coll.*, *29*(6), 153–154. Retrieved from http://dl.acm.org/citation.cfm?id=2602724.2602754

Edwards, S. H., Tilden, D. S., & Allevato, A. (2014). Pythy: Improving the introductory python programming experience. In *Proceedings of the 45th acm technical symposium on computer science education* (pp. 641–646). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2538862.2538977 doi: 10.1145/2538862.2538977

Enbody, R. J., & Punch, W. F. (2010). Performance of python cs1 students in mid-level non-python cs courses. In *Proceedings of the 41st acm technical symposium on computer science education* (pp. 520–523). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1734263.1734437 doi: 10.1145/1734263.1734437

Ericsson, K., & Simon, H. (1993). *Protocol analysis: Verbal reports as data (revised edition).* MIT Press, Cambridge, MA.

Fleury, A. E. (1991). Parameter passing: The rules the students construct. In *Proceedings of the twenty-second sigcse technical symposium on computer science education* (pp. 283–286). New York, NY, USA: ACM.

Goldwasser, M. H., & Letscher, D. (2008). Teaching an object-oriented cs1-: with python. In *Acm sigcse bulletin* (Vol. 40, pp. 42–46).

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. *SIGCSE Bull.*, *29*(1), 131–134.

Liberman, N., Beeri, C., & Ben-David Kolikant, Y. (2011). Difficulties in learning inheritance and polymorphism. *ACM Transactions on Computing Education (TOCE)*, *11*(1), 4.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., ... Whalley, J. L. (2006). Research perspectives on the objects-early debate. In *Working group reports on iticse on innovation and technology in computer science education* (pp. 146–165). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1189215.1189183` doi: 10.1145/1189215.1189183

Madison, S., & Gifford, J. (2002). Modular programming: novice misconceptions. *Journal of Research on Technology in Education*, *34*(3), 217–229.

Meyer, J. H. F., & Land, R. (2003). Threshold concepts and troublesome knowledge: linkages to ways of thinking and practising within the disciplines. In C. Rust (Ed.), *Improving student learning—ten years on.* OCSLD, Oxford.

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

Newhall, T., Meeden, L., Danner, A., Soni, A., Ruiz, F., & Wicentowski, R. (2014). A support program for introductory cs courses that improves student performance and retains students from underrepresented groups. In *Proceedings of the 45th acm technical symposium on computer science education* (pp. 433–438). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2538862.2538923` doi: 10.1145/2538862.2538923

Perkovic, L. (2011). *Introduction to computing using python: An application development focus.* Wiley Publishing, Hoboken, NJ.

Sajaniemi, J., Kuittinen, M., & Tikansalo, T. (2008, January). A study of the development of students' visualizations of program state during an elementary object-oriented programming course. *J. Educ. Resour. Comput.*, *7*(4), 3:1–3:31. Retrieved from `http://doi.acm.org/10.1145/1316450.1316453` doi: 10.1145/1316450.1316453

Sanders, K., & Thomas, L. (2007). Checklists for grading object-oriented cs1 programs: Concepts and misconceptions. In *Proceedings of the 12th annual sigcse conference on innovation and technology in computer science education* (pp. 166–170). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1268784.1268834` doi: 10.1145/1268784.1268834

Sauro, J., & Lewis, J. R. (2012). *Quantifying the user experience: Practical statistics for user research.* Elsevier.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *Software Engineering, IEEE Transactions on*(5), 595–609.

Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on innovation & technology in computer science education* (pp. 39–44). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2591708.2591749` doi: 10.1145/2591708.2591749