PyCon 2008: Presentation Proposal

Using Python To Teach Object-Oriented Programming in CS1

Michael H. Goldwasser David Letscher

Dept. Mathematics and Computer Science
Saint Louis University
221 North Grand Blvd
St. Louis, Missouri 63103-2007

Summary:

In recent years, Python has made great inroads as an introductory language in computer science education, but few emphasize its object-oriented nature; it is most often introduced with a procedural paradigm. For those teaching object orientation in CS1, Java remains the predominant language.

We suggest that Python is an excellent choice for teaching an object-oriented CS1. Based on our experiences teaching such a course for three years and authoring a textbook "Object-Oriented Programming in Python" (`http://prenhall.com/goldwasser`), we discuss the issues involved in adopting Python for an object-oriented CS1, and the downstream effects on the remaining curriculum.

1 Introduction

Python use in computer science education has grown rapidly in recent years. The attraction generally stems from its clean and simple syntax, allowing students to devote greater effort toward learning conceptual issues and developing applications, with less emphasis on extraneous syntactical details. The lower overhead of Python can be leveraged in many ways. Some have used it to provide a gentler introduction to K–12 or non-majors [8, 18], or to support a brief programming unit of a larger breadth-first course [31]. Others use the newfound freedom to explore a specific application area (e.g., robotics [5], multimedia [16], scientific computing [12, 23, 30]).

Our goal is to advocate for Python as a wonderful teaching language for those wishing to emphasize object-oriented principles in an introductory programming course. At Saint Louis University, we began offering a redesigned computer science curriculum in Fall of 2005 using Python as the language for an object-oriented introduction to programming in CS1. We describe how this language choice affects the presentation of material when compared to the more mainstream view of object orientation in Java or C++. We also discuss how Python's use in CS1 affects the broader computer science curriculum.

2 The Debate on Object Orientation for CS1

While it is widely accepted that object-oriented principles are a necessary component of a computer science curriculum, the educational community has engaged in a debate for more than a decade as to when and how these principles should be introduced [3, 6, 11]. On one side of the debate, the concerns are that object-oriented principles are too complex for beginning students to grasp, and that teaching those principles takes time away from the basics that are needed in CS1 [4, 11, 25]. Others recommends an "objects-first" approach, described by the CC2001 Computer Science volume [19] as one that "emphasizes the principles of object-oriented programming and design from the very beginning." The presumed benefit is that it allows students to develop a mind-set in which object-oriented concepts have a natural place, rather than seeing object-oriented programming as a dramatic paradigm shift later in a curriculum [2, 9, 26, 32].

Proponents on both sides seem to agree that the development of a coherent object-oriented approach is challenging. The question arises as to whether the difficulties are inherent to teaching the paradigm, or a reflection on the education community's failure to develop a means to the end. A significant aspect of the challenge is finding a programming language and environment that supports the pedagogy. When discussing the objects-first curriculum, the CC2001 report warns:

> "Many of the languages used for object-oriented programming in industry — particularly C++, but to a certain extent Java as well — are significantly more complex than classical languages. Unless instructors take special care to introduce the material in a way that limits this complexity, such details can easily overwhelm introductory students."

Reports detail early students' struggles with both object-oriented principles and syntax [14, 21, 33]. To this end, many pedagogical tools have been developed for use in the classroom, such as BlueJ [20], DrJava [1], Java Power Tools [24], objectdraw[7], and materials developed by the ACM Java Task Force [27, 28] which are described in their charter as "pedagogical resources that will make it easier to teach Java to first-year computing students without having those students overwhelmed by its complexity." While efforts such as these represent the hard work of dedicated educators, the sheer volume of such efforts and the need for pedagogical tools on top of a core language reflects upon shortcomings of Java and C++ in supporting the introductory programming course.

3 Choosing a Programming Language for CS1

Any change to the design of CS1 is a non-trivial undertaking for an institution with major impact on the overall curriculum. The faculty members must consider how CS1 prepares students for subsequent courses, whether those later courses must be redesigned given a change to CS1, and how the overall degree prepares students for their careers.

The goals of a computer science program typically transcend the precise choice of programming language. More fundamental issues, such as the debate on when and how object orientation should be introduced, are more primary concerns. The instructional language has significance as a vehicle for delivering the conceptual material. It also impacts the students' experiences and hence their perspectives. In this section, we discuss some important issues that arise regarding the choice of language.

"Real World" Appeal

The significance of a programming language in industry or the lack thereof is not directly relevant to whether it is effective as a teaching language. Yet there is an intangible benefit to using a language that students (and employers) view as practical. Several language favored by educators due to their simplicity (e.g., Pascal, Scheme), garner the negative label of "academic" languages. Industry appeal is certainly a factor that led to the widespread adoption of Java into the computer science curriculum in the past decade. While Python's prominence in industry still trails that of Java and C++, its significance clearly shields it from any such academic label.

Transition to Other Languages

Whatever programming language is used for the first course, students will eventually need exposure to many languages and paradigms as part of the larger curriculum. Educators must consider how students will transition from the first language to the second (and beyond). For those focusing early on object orientation there may be particular concern that Python's style is quite different from the more familiar appearance of Java and C++.

There exists another hurdle in advocating for Python over Java or C++, if the primary argument is that those other languages are overly complex and that Python is easier to learn. Some may believe that the difficulty of learning a language like Java provides greater reason for *starting* in the language, so that students will have more time to grow accustomed.

To counter this viewpoint, it is important to demonstrate that concepts introduced in Python can be easily transferable to other languages. We address this specific issue in Section 5. The transition to Java and C++ can be eased by Python's support of cross-language integration, for example using Jython to access Java's AWT, or Boost for interacting with C/C++ code. The multi-paradigm nature of Python also allows it to bridge the gap to non-object-oriented paradigms (e.g., functional) in later courses.

Selecting a Textbook

A pragmatic issue in developing a CS1 offering is having an appropriate textbook or other such reading material for students. Fortunately, the growing popularity of Python has led to the development of several introductory textbooks [13, 17, 22, 34]. Yet for those wishing to emphasize object-oriented principles early in a course, the selection is rather limiting. Most establish a procedural style, introducing the concepts of object orientation as a later topic. For example, Zelle's popular CS1 text [34] does not introduce the syntax for a user-defined class until Chapter 10 of 13. Furthermore, its earlier treatment is often counter to the norms of object-oriented programming, such as use of the deprecated syntax `string.count(dna,'C')` rather than the usual `dna.count('C')`. Downey *et al.*'s "How to Think Like a Computer Scientist" [13] does not introduce the word "class" (other than to designate is as a keyword of the language) until Chapter 12 of 20.

These issues led us to author a more object-oriented Python text [15]. We hope that our contribution broadens the support for others and we expect that the range of available Python books will continue to broaden as more schools adopt the language.

4 Python's Impact on an Object-Oriented CS1

Our general reasons for using Python are quite similar to those who have used Python for a more procedural approach: to allow for greater emphasis on core principles with less unwanted focus on syntax. Our goal is a steady progression in which each lesson is consistent with previous lessons. In particular, we want the awareness of objects to be clear from the beginning, rather than an awkward paradigm shift later in the course. Yet, we do not want to front-load the course with every aspect of object-oriented programming, nor unnecessary details of language syntax. It is in this regard that Python's simplicity provides great advantage over the use of Java or C++.

Getting Started by Using Objects (+1)

We consider an "objects-first" curriculum to be one in which students are presented with a clear *awareness* of objects from the beginning of the course. However, this does not mean that we expect students to immediately *implement* their own classes from the beginning. Instead, a natural starting point is to allow students to manipulate objects from existing classes. This allows them to connect the method calling syntax to an underlying semantics, and to explore distinctions such as that between mutable and immutable objects.

Yet a typical challenge for an instructor is providing students with beginning examples of intuitive, tangible objects with which to interact. Often, this barrier is overcome by having students use packages that are not part of the standard language, such as microworlds (e.g., [10]) or graphics (e.g., [7, 24, 28, 29]). However, an over-reliance on classes that are not part of the standard language decreases the transparency and portability of a curriculum. Python supports a rich set of built-in classes (e.g., str, list, dict, file), providing immediate examples of both mutable and immutable objects with which to experiment.

A second challenge is how to support students writing their first programs without being overwhelmed by syntax. The requirement for a main routine in Java and C++, and worse yet the need for that routine to be a static member of a class in Java, causes students to be immediately exposed to aspects of a language that are not appropriate at an early stage of a course (the classic "public static void main" issue). To remedy this, pedagogical IDEs such as DrJava [1] and BlueJ [20] provide students with an interactive environment in which they can experiment with objects. DrJava supports an interactive session in which Java commands can be individually evaluated. BlueJ supports the concept of an "object bench" allowing users to instantiate and manipulate objects outside of the context of a typical program.

In Python, the standard interpreter can be used as just such a pedagogical tool. Our students' first computing experience in CS1 is an interactive session in the Python interpreter. For example, the following Python session shows a sample interaction with an instance of the **list** class.

```
>>> groceries = list()
>>> groceries.append('bread')
>>> groceries.append('milk')
>>> groceries
['bread', 'milk']
>>> groceries.append('cereal')
>>> groceries.sort()
>>> groceries
['bread', 'cereal', 'milk']
```

As with DrJava, each command is evaluated when entered, allowing the user to observe the effect of the commands (or to be informed in case of an error). The interpreter is also similar to BlueJ's object bench, as it allows for the instantiation and manipulation of objects without the syntactic overhead of a formal main routine. Yet unlike DrJava and BlueJ, the Python interpreter is part of the standard distribution; we avoid an early reliance on external tools. Our students' transition from interactive sessions to writing source code is rather straightforward. The source code is a script, written with almost precisely the same keystrokes that were otherwise typed directly into the interpreter. The only new lesson at this time is a focus on the distinction between the role of the software's developer versus its user. This leads to the introduction of the **print** and **raw_input** commands for user interactions.

A Consistent Object Model (+1)

A clear advantage of Python is its consistent object model. All data types are classes, in contrast to the discord between Java's primitive types and object types. Furthermore, each identifier serves as a reference to an underlying object (akin to the treatment of Java's object types). This is the only model in Python, unlike Java which supports value and reference variables, and C++ which allows value, reference or pointer models.

The consistency carries over to the assignment semantics. An identifier can be portrayed as a label referencing the underlying object, in which case the assignment statement associates (or reassociates) the identifier on the left-hand side as a label for the value expressed on the right-hand side. Parameter passing is explained with similar consistency; an identifier serving as a formal parameter is assigned to the underlying object designated as the actual parameter.

Dynamic Typing (+0)

For object-oriented purists are often concerned that Python is a dynamically-typed language. Perhaps this gives an initial (yet false) impression that data types are amorphous. This would seem counter to a focus on objects being drawn from classes. We emphasize that Python is both dynamically typed and *strongly* typed. Each object has a definitive type that never changes and self-awareness of its type encoded in its representation. There is simply no syntactic declaration of the data type associated with an *identifier*.

The interpreter can be used to reinforce this lesson. For example, the command type(groceries) reports the type of object[1] currently associated with the identifier groceries. A directory of the names of the object's members are reported by the command dir(groceries), and formal documentation is provided with the interpreter command help(groceries). To instill a deeper understanding of the relationship between an identifier and an object, the command id(groceries) reports a unique integer identifying the underlying object (in effect, a memory address).

Class Definition Syntax (+0)

The role of the **self** reference in a class definition is one of the more controversial issues (both within the Python community and for instructors who are new to Python). The **self** reference must be explicitly listed as a formal parameter in the signature, and explicitly qualified when accessing a

[1]Technically, this is only so for new-style classes. When called on an old-style instance, type reports `'instance'`.

member of the instance. While the general mechanism is similar in purpose to **this** in Java or C++, the usage in Python is in stark contrast to the standard practices in Java and C++. This tends to put off instructors or students who are new to Python.

We wish to address these two related points separately, beginning with the requirement that **self** be used as a qualifier when accessing a member of the instance. Although this is one way in which Python's syntax is more verbose than that of Java and C++, we consider this requirement to be a valuable asset when teaching object-oriented programming. The use of the **self** qualifier draws attention to the distinction between instance variables and local variables. This is already a major lesson that must be addressed when teaching object orientation, and a potential confusion with the implicitness of **this** in Java and C++ (or the masking of an instance variable by the errant declaration of a similarly named local variable). The use of the qualifier is more consistent with the standard syntax for accessing a member of an object. Consider the following excerpt from a Point class.

```
def distance(self, other):
    dx = self._x − other._x
    dy = self._y − other._y
    return math.sqrt(dx ∗ dx + dy ∗ dy)
```

Notice the nice symmetry between **self** and other. In this context, **self** and other are presumed to each reference a Point instance. We use the syntax **self**._x just as we use the syntax other._x. In contrast, notice that the (unqualified) identifier dx is local to the function body. The requirement for using a **self** qualifier is also easy to motivate. Since Python is dynamically typed, the data members of a class are not formally declared. It is therefore impossible for the system to otherwise discern which unqualified names we intended as members and which as local variables.

The issue of **self** appearing explicitly on the parameter list is more bothersome in the context of CS1. It would be easier at this stage if **self** were a keyword of the language and *implicit* in the signature (although we recognize its purpose in the more general language syntax). The biggest problem with its explicit appearance in the method definition is the unusual discrepancy it creates between the signature as seen by the caller versus the signature as seen be the callee. This issue not only effects students when defining classes, it even taints their view when using classes. To illustrate the problem, consider executing the following simple script.

```
class OurList:
    def append(self, element):
        pass
data = OurList( )
data.append( )
```

The final command results in the following misleading error message:

```
TypeError: append() takes exactly 2 arguments (1 given)
```

The issue is that as a function, OurList.append expects two parameters, and the syntax data.append( ) is equivalent to OurList.append(data), thus only specifying the first. Yet since the exception is passed back to the caller, this seems an inappropriate message in this scenario. It certainly will confuse a beginning students. The developers of Python seem to recognize that this is a misleading error message for such a call. When working with the built-in list class, the same errant call data.append( ) produces the following message.

```
TypeError: append() takes exactly one argument (0 given)
```

This is a legitimately informative message informing the caller that an expected parameter was omitted; yet this is a behavior that cannot easily be replicated for non-built-in classes.

One other concern is that Python allows techniques that are atypical for object-oriented programming. Most notably, members can be added or removed from an instance's namespace at any point in time (not just during initialization). Furthermore, it is possible to have two instances form the same class supporting a different set of members. For CS1, we model a more standard view of object-orientation, introducing all necessary members uniformly within the __init__ method.

Encapsulation (-1)

Python does not contain any syntactic declarations of visibility for members of a class (i.e., **public**, **private**, **protected**). Nor does Python strictly enforce any notion of access control. This style is consistent with Python's general "we're all adults" philosophy, yet it causes discomfort for object-oriented purists, as encapsulation is a core principle.

Although we generally are in favor of Python's reduced syntax, the designation and enforcement of access control in Java and C++ is not an undue burden. That said, we certainly teach the concept of encapsulation and establish it as de facto practice in all of our own examples.

Furthermore, Python has *informal* support for encapsulation based on naming conventions. Elements that are identified starting with a single underscore (e.g., _x) are considered to be non-public by convention. While they can still be accessed directly from other components, their use is discouraged. Python makes some effort to hide such elements, as they are not automatically displayed when using the help command, nor are they imported when using the **from** module **import** * syntax. Although Python also differentiates between identifiers starting with a single versus double underscore (akin to protected versus private), we do not bother introducing this to our students.

Generics and Polymorphism (+1)

One of the greatest advantages of Python's dynamic typing in an object-oriented context is the relative ease in accomplishing generic programming and polymorphism. For example, we already demonstrated the user of Python's **list** class at the beginning of a CS1 course. A list represents an ordered sequence of objects, internally implemented as an expandable array of references akin to Java's `ArrayList` class. While the concept of a container is similar in the languages, the simplicity of Python's syntax is in stark contrast to the overhead when downcasting from Java's Object class in Java 1.4, or using parameterized types in Java 5 and C++.

Python's dynamic typing also eases the introduction to polymorphism, providing yet another opportunity to focus on an important object-oriented concept with minimal syntactic overhead. When a programmer accesses a method of an object with a syntax foo.bar( ), the resolution of bar is performed at run time. If the object identified as foo has such a method, all is well; otherwise an exception is raised. Consider the goal of writing a polymorphic function that accepts a parameter assumed to support a subset of behaviors. In Python, we simply use it. In Java, this task is accomplished by formally declaring a common **interface**, having all relevant classes formally declare their implementation of the interface, and defining the polymorphic function to accept a variable with the designated interface type. Pure abstract classes serve a similar purpose in C++.

Inheritance (0)

The use of inheritance is quite natural, with minimal syntactic overhead. The declaration for a class being defined from scratch begins with the syntax **class** Person**:** . A child class is declared as **class** Student(Person)**:** . Multiple inheritance is supported, as in **class** Button(Text, Rectangle)**:** .

Name resolution proceeds naturally, looking for members at the instance scope, then at the class scope, then the parent class scope, and so on. This provides a clear mechanism for overriding behaviors. An overridden member (e.g., a parent constructor) can be accessed with an explicit qualifier, as in Person.__init__ from within the Student class context.

5 Transition to Java and C++

We teach CS1 with the forethought of our students' later transition to other object-oriented programming languages, most immediately Java and C++. Although the languages differ, we do not have to unteach any of the concepts introduced in Python. We must simply augment those experiences with additional lessons. A certain amount of the transition is due to superficial differences, such as the designation of block structures, the names of primitive types, the use of I/O, and the role of a formal main routine. Of the more significant differences, 90% can be related to a single theme: *moving from an interpreted dynamically-typed language to one that is compiled and statically typed.* This affects the software development cycle as well as the language syntax. We motivate the switch pragmatically, noting that greater efficiency is possible if the system can perform more work at compile time and less at run time.

Explicit type declarations arise in the context of local variables, parameters, return values, and data members of a class. Our Python students are very aware of assumed data types in these contexts, so this is not a major hurdle. Other aspects of the transition are framed in this light. For example, with explicit declaration of a class's data members it becomes possible for the compiler to recognize $\_x$ as an instance variable without qualification of an explicit **self** reference. The need for additional syntax with generics and polymorphism in Java and C++, as discussed in Section 4, can be motivated by the need for compile-time checking. The declaration of formal access controls to enforce encapsulation is another example of more rigorous compile-time checking to support the principles of object orientation.

The varying object models of Java and C++ are novel for students transitioning from Python. The jump from Python to Java is more benign. Python's model for identifiers, assignment semantics, and information passing is identical to Java's reference model for object types. It is only the distinction made by Java for primitive types that must be explained. Both Python and Java rely on garbage collection to shield a programmer from low-level memory management. The jump from Python to C++ is more extreme due to the inherent complexity of C++. There are three different storage models (value, reference, pointer), and any of those can be selected for any particular piece of data. In addition, the distinction between static and dynamic memory allocation places a greater burden on the C++ programmer.

Our CS2 offering is a data structures course taught using C++. This course builds upon object-oriented principles in designing abstract data types while focusing on low-level efficiency of the encapsulated implementations. We intentionally chose C++ as the second language knowing that it was at the far extreme of Python on the spectrum of object-oriented languages. The focus on data structures and efficiency allows us to motivate the introduction of system-level concepts such

as pointers, memory management and varying models for information passing. We explain that, just as greater compile-time checking streamlines the run-time performance, the fine-tuned control afforded to a programmer in C++ offers an opportunity for greater efficiency (if used wisely). That ties in with the larger theme of our data structures course. Having made the transition from Python to C++, we introduce Java to our students as part of a CS3 course focused on object-oriented design. This context allows us to address aspects of the three languages that differ significantly, such as the underlying object model, use of inheritance, polymorphism, and generics.

6 Conclusions

We have been very pleased with the use of Python, finding that it affords a clear, coherent, and consistent presentation of object-oriented programming. Our initial batch of Python CS1 students are progressing through the curriculum, and anecdotally the change has been successful.

Yet we recognize that a widespread revision to CS1 is a major undertaking for an institution. At this point, our goal is to disseminate the strategy in hope of gaining a wider set of early adopters and building a community for discussion. In the long run, we would like to see formal studies as to the short-term and long-term effectiveness of the approach in shaping a student's educational experience.

References

[1] E. Allen, R. Cartwright, and B. Stoler. DrJava: A lightweight pedagogic environment for Java. In *Proc. 33rd SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 137–141, Covington, Kentucky, Feb. 27–Mar. 3, 2002.

[2] C. Alphonce and P. Ventura. Object orientation in CS1-CS2 by design. In *Proc. Seventh Annual Conf. on Innovation and Technology in Computer Science (ITiCSE)*, pages 70–74, Aarhus, Denmark, June 2002.

[3] O. Astrachan, K. Bruce, , E. Koffman, M. Kölling, and S. Reges. Resolved: Objects early has failed. In *Proc. 36th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 451–452, St. Louis, Missouri, Feb. 2005. ACM Press.

[4] F. Bailie, M. Courtney, K. Murray, R. Schiaffino, and S. Tuohy. Objects first – does it work? *J. Computing Sciences in Colleges*, 19(2):303–305, Dec. 2003.

[5] D. Blank, L. Meedan, and D. Kumar. Python robotics: An environment for exploring robotics beyond LEGOs. In *Proc. 34th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 317–321, Reno, Nevada, Feb. 2003.

[6] K. B. Bruce. Controversy on how to teach CS 1: A discusion on the SIGCSE-members mailing list. *SIGCSE Bulletin*, 37(2):111–116, June 2005.

[7] K. B. Bruce, A. Danyluk, and T. Murtaugh. A library to support a graphics-based object-first approach to CS 1. In *Proc. 32nd SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 6–10, Charlotte, North Carolina, Feb. 2001.

[8] V. Ceder. Good-bye hello world: Rethinking teaching with Python. In *PyCon 2007*, Addison, TX, Feb. 2007.

[9] S. Cooper, W. Dann, and R. Pausch. Teaching objects-first in introductory computer science. In *Proc. 34th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 191–195, Reno, Nevada, Feb. 2003.

[10] W. Dann, S. Cooper, and R. Pausch. *Learing to Program with Alice*. Prentice Hall, 2006.

[11] R. Decker and S. Hirshfield. The top 10 reasons why object-oriented programming can't be taught in CS1. In *Proc. 25th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 51–55, Phoenix, Arizona, Mar. 1994.

[12] Z. Dodds, C. Alvarado, G. Kuenning, and R. Libeskind-Hadas. Breadth-first CS 1 for scientists. In *Proc. 12th Annual Conf. on Innovation and Technology in Computer Science (ITiCSE)*, pages 23–27, Dundee, Scotland, June 2007.

[13] A. B. Downey, J. Elkner, and C. Meyers. *How to Think Like a Computer Scientist: Learning with Python*. Green Tea Press, Needham, MA, 2002.

[14] A. E. Fleury. Programming in Java: Student-constructed rules. In *Proc. 31nd SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 197–201, Austin, Texas, May 2000.

[15] M. H. Goldwasser and D. Letscher. *Object-Oriented Programming in Python*. Prentice Hall, 2008.

[16] M. Guzdial. A media computation course for non-majors. In *Proc. Eighth Annual Conf. on Innovation and Technology in Computer Science (ITiCSE)*, pages 104–108, Thessaloniki, Greece, June 30–July 2, 2003.

[17] M. Guzdial. *Introduction to Computing and Programming in Python: A Multimedia Approach*. Prentice Hall, 2005.

[18] A. Harrington. Python for students of the modern world. In *PyCon 2007*, Addison, TX, Feb. 2007.

[19] Joint Task Force on Computing Curricula. *Computing Curricula 2001: Computer Science Final Report*. IEEE Computer Society and the Association for Computing Machinery, Dec. 2001. http://www.computer.org/education/cc2001/final.

[20] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *J. Computer Science Education*, 4(13):249–268, Dec. 2004.

[21] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. Early programming: A study of the difficulties of novice programmers. In *Proc. Tenth Annual Conf. on Innovation and Technology in Computer Science (ITiCSE)*, pages 14–18, Monte da Caparica, Portugal, June 2005.

[22] B. N. Miller and D. L. Ranum. *Problem Solving with Algorithms and Data Structures Using Python*. Franklin Beedle & Associates, 2006.

[23] R. P. Olenick. Visual Python in a computational physics course. In *PyCon 2007*, Addison, TX, Feb. 2007.

[24] R. Rasala, J. Raab, and V. K. Proulx. Java Power Tools: Model software for teaching object-oriented design. In *Proc. 32nd SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 297–301, Charlotte, North Carolina, Feb. 2001.

[25] S. Reges. Back to basics in CS1 and CS2. In *Proc. 37th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 293–297, Houston, Texas, Mar. 2006.

[26] R. J. Reid. The object oriented paradigm in CS1. In *Proc. 24th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 265–269, Indianapolis, Indiana, Feb. 1993.

[27] E. Roberts. The dream of a common language: The search for simplicity and stability in computer science education. In *Proc. 35th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 115–119, Norfolk, Virginia, Mar. 2004. ACM Press.

[28] E. Roberts, K. Bruce, R. Cutler, J. H. Cross II, S. Grissom, K. Klee, S. Rodger, F. Trees, I. Utting, and F. Yellin. The ACM Java task force: Final report. In *Proc. 37th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 131–132, Houston, Texas, Mar. 2006.

[29] E. Roberts and A. Picard. Designing a Java graphics library for CS1. In *Proc. Third Annual Conf. on Innovation and Technology in Computer Science (ITiCSE)*, pages 213–218, Dublin, Ireland, Aug. 1998.

[30] D. Scherer, P. Dubois, and B. Sherwood. VPython: 3D interactive scientific graphics for students. *Computing in Science & Engineering*, 2(5):56–62, Sept./Oct. 2000.

[31] C. Shannon. Another breadth-first approach to CS I uisng Python. In *Proc. 34th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 248–251, Reno, Nevada, Feb. 2003.

[32] M. C. Temte. Let's begin introducing the object-oriented paradigm. In *Proc. 22nd SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 73–77, San Antonio, Texas, Mar. 1991.

[33] I. Utting. Problems in the initial teaching of programming using Java: The case for replacing J2SE with J2ME. In *Proc. 11th Annual Conf. on Innovation and Technology in Computer Science (ITiCSE)*, pages 193–196, Bologna, Italy, June 2006.

[34] J. M. Zelle. *Python Programming: An Introduction to Computer Science*. Franklin Beedle & Associates, 2003.