# Teaching an Object-Oriented CS1 — with Python

Michael H. Goldwasser
Dept. Mathematics and Computer Science
Saint Louis University
220 North Grand Blvd
St. Louis, Missouri 63103-2007
goldwamh@slu.edu

David Letscher
Dept. Mathematics and Computer Science
Saint Louis University
220 North Grand Blvd
St. Louis, Missouri 63103-2007
letscher@slu.edu

## ABSTRACT

There is an ongoing debate regarding the role of object orientation in the introductory programming sequence. While the pendulum swings to and fro between the "objects first" and "back to basics" extremes, there is general agreement that object-oriented programming is central to modern software development and therefore integral to a computer science curriculum. Developing effective approaches to teach these principles raises challenges that have been exacerbated by the use of Java or C++ as the first instructional language.

In this paper, we recommend Python as an excellent choice for teaching an object-oriented CS1. Although often viewed as a "scripting" language, Python is a fully object-oriented language with a consistent object model and a rich set of built-in classes. Based upon our experiences, we describe aspects of the language that help support a balanced introduction to object orientation in CS1. We also discuss the downstream effects on our students' transition to Java and C++ in subsequent courses.

## Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.2 [**Programming Languages**]: Language Classifications—*object-oriented languages, Python*

## General Terms

Languages

## Keywords

Python, Object Orientation, CS1

## 1. INTRODUCTION

While it is widely accepted that object-oriented principles are a necessary component of a computer science curriculum, the community has engaged in a debate for more than a decade as to when and how these principles should be introduced [2, 5, 9]. One approach suggested by the CC2001 Computer Science volume [16] is an objects-first paradigm that "emphasizes the principles of object-oriented programming and design from the very beginning." The presumed benefit is that it allow students to develop a mindset where object-oriented concepts have a natural place, rather than seeing object-oriented programming as a dramatic paradigm shift later in a curriculum [7, 24, 27].

On the other 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 [3, 23]. Proponents on all 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 challenge is finding a choice of programming language and environment that is supportive rather than detrimental. When discussing the objects-first curriculum, the CC2001 report warns that "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."

Educators have reported on beginning students' struggles with both object-oriented principles and syntax [12, 18, 28]. For this reason, many pedagogical tools have been developed for use in the classroom, such as BlueJ [17], DrJava [1], Java Power Tools [22], objectdraw[6]. Materials developed by the ACM Java Task Force [25] 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 an introductory programming course.

In this paper, we advocate for Python as the instructional language for an object-oriented CS1. An overview of the existing use of Python in education is given in Section 2. Our main contribution, in Section 3, is a pedagogy for introducing object orientation with Python. Our curriculum and the subsequent transition to Java and C++ is discussed in Section 4. We conclude in Section 5.


*ITiCSE'08,* June 30–July 2, 2008, Madrid, Spain.

## 2. PYTHON IN EDUCATION

Python's use in academia has grown rapidly in recent years [4, 10, 20, 21, 26], leading to the development of several introductory textbooks [11, 13, 14, 15, 29]. 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. Python is also widely used in industry and championed by several prominent companies. This shields it from the unfortunate "academic" label sometimes associated with other instructional languages favored for simplicity (e.g., Pascal, Scheme).

The novelty of our approach [14] is the strong emphasis given to Python's object orientation. Others have leveraged the lower overhead of Python to provide a gentler introduction to non-majors, to support a brief programming unit of a larger breadth-first course, or to explore a specific application area (e.g., robotics, multimedia, scientific computing). Yet these existing approaches greatly delay the treatment of object-oriented principles. As a benchmark, we consider the placement of user-defined classes in several popular Python textbooks. This topic is introduced in Chapter 10 of 13 within [29], Chapter 14 of 16 within [15], Chapter 9 of 12 within [13], and Chapter 12 of 20 within [11] (and slated to slip to the appendix in the second edition).

## 3. PYTHON'S IMPACT ON TEACHING OBJECT ORIENTATION IN CS1

Our general reason for using Python is quite similar to those using Python for teaching a procedural approach: to allow 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++. In the remainder of this section, we outline the impact of the language choice, paying particular attention to oft-stated challenges associated with teaching an objects-first curriculum.

### 3.1 Getting Started by Using Objects

An important first step when introducing object orientation is providing students with 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., [8]) or graphics (e.g., [6, 22, 25]). 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 with 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 [17] 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']
>>> groceries.pop()
'milk'
>>> groceries
['bread', 'cereal']
```

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.

### 3.2 A Consistent Object Model

Another advantage of Python is a 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, and 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.

## 3.3 Dynamic Typing

The lack of explicit type declaration in Python greatly simplifies its syntax. This allows a programmer to begin using an object with a command such as w = Widget( ), rather than the more verbose Widget w = new Widget( ) as seen in Java. Dynamic typing similarly streamlines the declaration of function signatures and, as we will discuss in Section 3.6, the support for generics and polymorphism.

Admittedly, object-oriented purists voice concern about using a dynamically-typed language. Perhaps this gives an initial (yet false) impression that data types are amorphous. This might 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 currently associated with the identifier groceries. The names of the object's methods 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).

## 3.4 Class Definitions

Once students have been introduced to the use of objects and the basic control structures, the high-level syntactic structure of a Python class definition follows a familiar style. As an example, Figure 1 gives a partial definition of a Point class. While this syntax may seem unusual to experienced Java and C++ programmers, it is quite natural for students learning Python. There is an initial declaration followed by an indented block of code, with individual methods defined using a syntax similar to stand-alone functions.

As an artifact of dynamic typing, there are no explicit declaration of the instance variables; they are introduced on the fly, typically from within the body of an initializer method, as shown in lines 2–4 of Figure 1. A related issue is the use of parameter self as a reference to the particular instance upon which a method is invoked (akin to the implicit **this** of Java or C++). An external call of p.scale(2) is internally equivalent to the signature scale(p, 2). The self reference is used to explicitly qualify all members of the object, such as the data member self._x at line 3 and the intermediate call to member function scale at line 26. This syntax distinguishes members from unqualified identifiers, such as the local variable dx at line 19.

While this is one way in which Python's syntax is more verbose than that of Java and C++, the use of the self qualifier draws attention to the distinction between instance scope versus local scope. This is already a major theme 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). Notice the parallel between self and other in the distance method. In this context, self is a reference to a Point as is parameter other (presumably). At line 19, we use the syntax self._x just as we use the syntax other._x.

```
1  class Point:
2    def __init__(self):
3      self._x = 0
4      self._y = 0
5
6    def getX(self):
7      return self._x
8
9    def setX(self, x):
10     self._x = x
11
12   # ... some methods omitted for brevity ...
13
14   def scale(self, factor):
15     self._x *= factor
16     self._y *= factor
17
18   def distance(self, other):
19     dx = self._x - other._x
20     dy = self._y - other._y
21     return sqrt(dx * dx + dy * dy)
22
23   def normalize(self):
24     mag = self.distance( Point( ) )
25     if mag > 0:
26       self.scale(1/mag)
```

**Figure 1: A partial definition of a Point class.**

## 3.5 Encapsulation

A Python class definition does not contain any syntactic declarations of visibility (i.e., **public**, **private**, **protected**). Nor does Python strictly enforce any access control. Again, this causes discomfort for object-oriented purists, as formal access control is an important tool for software developers.

Yet in the context of CS1, students do not recognize this as an issue. We teach encapsulation as a core principle of object-oriented programming, and establish it as de facto practice in all examples. We simply do not introduce the syntax of access control.

Python has *informal* support for encapsulation based on naming conventions. Elements having identifiers starting with a single underscore (e.g., _x) are considered non-public by convention. While they can still be accessed directly from other components, such 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.

## 3.6 Generics and Polymorphism

One of the great advantages of Python's dynamic typing in an object-oriented context is the relative ease in accomplishing generic programming and polymorphism. We previously demonstrated the ease of use of Python's **list** class for beginners. A list represents an ordered sequence of generic 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. This style of type checking is often termed *duck typing* (if it walks like a duck and quacks like a duck, it must be a duck).

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++.

## 3.7 Inheritance

Python supports single and multiple inheritance with minimal syntactic overhead. The name of any parent classes are specified within parentheses as part of the child class declaration. 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 using the name of the parent class as an explicit qualifier for the method.

As an example, Figure 2 defines a child class patterned after the Java implementation of `Dictionary2.java` given by Lewis and Loftus [19]. Note the declaration of Book2 as a parent class in line 1 and the invocation of the parent constructor at line 3.

# 4. TRANSITION TO JAVA AND C++

We teach CS1 with the forethought of our students' later transition to other object-oriented programming languages, most notably Java and C++. The concepts introduced in Python do not need to be untaught; we 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. Ninety percent of the more significant differences are related to a single theme: moving from an interpreted dynamically-typed language to one that is compiled and statically-typed. This affects the syntax and the software development cycle. We motivate the switch pragmatically, noting that greater run-time efficiency is achieved when the system performs more work at compile time.

```
1  class Dictionary2(Book2):
2    def __init__(self, numPages, numDefinitions):
3      Book2.__init__(self, numPages) # parent version
4      self._definitions = numDefinitions # new attribute
5
6    def computeRatio(self):
7      return self._definitions/self._pages
8
9    def setDefinitions(self, numDefinitions):
10     self._definitions = numDefinitions
11
12   def getDefinitions(self):
13     return self._definitions
```

**Figure 2: Python variant of the Dictionary2 class, originally given in Java by Lewis and Loftus [18].**

Explicit type declarations arise in the context of local variables, parameters, return values, and data members of a class. Our Python students are keenly aware of (presumed) 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 members it becomes possible for the compiler to recognize _x as an instance variable without an explicit self qualifier. The need for additional syntax with generics and polymorphism in Java and C++ 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 greater responsibility on the C++ programmer.

In our curriculum, CS2 is a traditional data structures course building upon object-oriented principles in designing abstract data types with efficient low-level implementations encapsulated within. We intentionally transition to C++ for this course, knowing that it is 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. 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.

# 5. 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. 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. 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.

## 6. 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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] T. Gaddis. *Starting Out with Python*. Addison-Wesley, 2009.

[14] M. H. Goldwasser and D. Letscher. *Object-Oriented Programming in Python*. Prentice Hall, 2007.

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

[16] 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.

[17] 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.

[18] 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.

[19] J. Lewis and W. Loftus. *Java Software Solutions: Foundations of Program Design*. Addison-Wesley, fifth edition, 2007.

[20] A. Radenski. "Python first": A lab-based digital introduction to computer science. In *Proc. 11th Annual Conf. on Innovation and Technology in Computer Science (ITiCSE)*, pages 197–201, Bologna, Italy, June 2006.

[21] D. Ranum, B. Miller, J. Zelle, and M. Guzdial. Successful approaches to teaching introductory computer science courses with Python. In *Proc. 37th SIGCSE Technical Symp. on Computer Science Education (SIGCSE)*, pages 396–397, Houston, Texas, Mar. 2006.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] M. 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.

[28] 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.

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