Teaching an Object-Oriented CS1 in Python

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

Abstract

Python’s use in education has grown rapidly due to its elegantly simple syntax. Though often viewed as a “scripting language,” Python is a fully object-oriented language with an extremely consistent object model and a rich set of built-in classes. We are finishing our second year using Python as the language for an object-oriented CS1 course. Based on these experiences, we discuss aspects of Python which make it a particularly attractive language for such a course design.

We frame our formal analysis of Python’s suitability using criteria established by Kölling as requirements for a first-year object-oriented teaching language. Then we describe our personal experiences and strategies when using Python in CS1 followed by a mainstream object-oriented language in CS2.

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

Python’s use in education has been rapidly growing, both within the computer science curriculum and more broadly in other sciences [9, 12, 21, 35, 43, 44]. This has led to the development of several introductory textbooks using Python [14, 19, 22, 31, 46]. The initial attraction to Python centers on its clean and simple syntax, allowing students to focus greater efforts on conceptual issues and less on extraneous syntactical details. Much of the existing curriculum development has centered around the imperative programming paradigm, perhaps stemming from Python’s reputation as a popular scripting language. Our goal is to highlight Python as an extremely attractive choice when teaching an object-oriented introduction.

Teaching Language for the First Course. In promoting the use of Python for CS1, we are in no way suggesting that other languages be marginalized from the larger computer science curriculum. The issue at hand is which language best serves the goals of a fundamental, introductory course. To frame our own evaluation, we consider a 1999 paper [25] in which Kölling proposes the following high-level criteria as “requirements for an object-oriented language ... in the context of the use as a teaching language for beginners.”

1. Readable syntax
2. Simple object/execution model
3. Suitable environment
4. High level
5. No redundancy
6. Small
7. Pure object-orientation
8. Clean concepts
9. Safety
10. Support for correctness assurance
11. Easy transition to other languages

Kölling’s paper describes these criteria in detail and continues with an explicit evaluation of four candidate languages: C++, Java, Eiffel and Smalltalk. In summary, he concludes that C++ fails to meet almost all requirements. Java is viewed as better than C++ but not a final solution to all the problems. He writes positively of Smalltalk though with some reservations, most notably its lack of readability, its lack of static typing and the reliance upon a large library of classes. Eiffel receives the highest praise, though with the fatal drawback being the lack of a simple and widely available environment.

[Copyright notice will appear here once ‘preprint’ option is removed.]
**Predominance of Java.** During the past decade, Java has become the most widely used language for instruction in CS1, surpassing the use of C++ and non-object-oriented languages such as C or Scheme. It has also been adopted as the language for the AP curriculum. Its dominance in education is due to a combination of factors. Java’s prominence in industry skyrocketed with its support of portable, web applications. This created pressure upon academia to prepare students for use of the language. At the same time, educators had a broader interest in incorporating object-oriented principles earlier in the curriculum. Kölling writes, “When only compared to C++, Java appears like the saviour we’ve been waiting for. When evaluating against our requirements, however, Java is far from being an ideal solution.”

In practice, managing the use of Java at an introductory level has proved to be quite challenging. In its Computer Science Final Report [24], the ACM/IEEE Joint Task Force on Computing Curriculum 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.”

Indeed, reports describe beginning students’ struggles with object-oriented principles and language syntax [15, 16, 30]. A cottage industry has grown as a result, with various educators offering their own strategies or tools for using Java in the introductory course. These include BlueJ [11, 27, 32], DrJava [10, 23], Java Power Tools [33, 36], objectdraw [13], and tools from the ACM Java Task Force [39, 40, 41] among others [17, 18, 20, 28, 29, 34, 37, 38, 45].

Many of those efforts represent the hard work of dedicated educators in developing a working strategy based on the use of Java. However the sheer volume of such efforts and the need for pedagogical tools on top of the core language reflects upon shortcomings of Java in supporting an introductory pedagogy. The community might be better served by exploring alternatives for the first course. In this regard, we echo a statement of Kölling from 1999 (applicable to Java as well as C++):

> “We have to teach our students to program in a language that is relevant to industry. But it is a wrong conclusion that this means that we have to start teaching in C++. Most people would agree that, by the time students leave a university, they should be competent programmers in an industry-relevant language. This only means that students should end up with strong, say C++, skills, but not that they must start with it. On the contrary: We as teachers, have to ensure that students learn *programming* as opposed to a *programming language.*”

**Our Curriculum.** We redeveloped our entire computer science curriculum in 2004-2005 and deployed the new introductory programming sequence beginning in Fall 2005. Our “CS1” course is a four-credit, object-oriented introduction using Python as the instructional language. The course instills an immediate awareness of object-oriented principles through the manipulation of built-in objects. After some time exploring those existing object, together with standard control structures, we progress to the design and development of user-defined classes, software engineering principles, use of inheritance and several advanced topics.

Our “CS2” offering is a four-credit, 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 in 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++, Java falls nicely in the middle. 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 which are quite different, such as the underlying object-model, use of inheritance and generics.

**Organization of this Presentation.** The remainder of this paper is organized in two major portions. We begin in Section 2 by evaluating the Python language features in the context of Kölling’s formal criteria for teaching object-oriented programming. In Section 3, we discuss more informally our own strategies when using Python in CS1, together with the transition to another language in CS2 and beyond. We conclude in Section 4.

## 2. Evaluating Python via Kölling’s Criteria

We use Kölling’s framework to evaluate the appropriateness of Python for teaching an object-oriented introduction. By no means are we suggesting that Python is perfect on all accounts, and we try to highlight the pros and the cons. By Kölling’s own acknowledgment, some of the stated requirements contradict each other. “For a good teaching language it will be essential to strike a good balance between these conflicting goals.”

### 2.1 Readable syntax

This may be the greatest single strength of Python, and the reason for its popularity both in object-oriented programming and other paradigms. Commands can be terminated by a newline character rather than punctuation (e.g., ;); nesting of structures is accomplished through indentation rather than punctuation (e.g., { }).

---

*Teaching an Object-Oriented CS1 in Python*
The language uses keywords **and**, **or** and **not** rather than symbols &&, || and !. The built-in containers support legible syntaxes such as `val in data` or `val not in data`, and these can be overloaded for user-defined containers by implementing a special method, `__contains__`. As a result, Python source code is typically very legible, even to non-Python programmers.

As a more general issue, Kölling suggests that “the same syntax should be used for same semantics, different syntax for different semantics.” Overall, Python does well in this respect, but there are a few inconsistencies. Most notably, there are some built-in functions that use a calling syntax which is atypical of object-oriented form, e.g., `len(data)` rather than `data.len( ); max(data)` rather than `data.max( )`.

There is also a discrepancy between the signature of a method from the caller’s perspective (e.g., `thing.foo(x)` versus the declaration made from the callee’s perspective (e.g., `def foo(self, val)`). From the caller’s perspective it has one parameter yet from the callee’s it has two parameters. We will address this aspect in Section 2.4.

### 2.2 Simple object/execution model

This too is a clear strength of Python. All types of data, including primitives, are treated as first-class objects. There is a single object model, namely that identifiers reference the underlying object (akin to treatment of Java’s object types). There is no way to interact with an object other than through use of an established identifier.

Assignment semantics is treated consistently as well, with the expression `lhs = rhs` serving to (re)assign the identifier on the left to a reference to the object expressed on the right. Parameter passing is also consistent, with the identifiers serving as formal parameters associated with the expressed actual parameters.

This consistency is in stark contrast to Java’s distinction between primitive types and object types. Consider for example the inconsistency of the following code in Java.

```java
int i, j;
Widget x, y;
```

Note that first statement results in the creation of two integers, whereas the second statement results in the creation of two Widget references (yet no actual Widget instances). This inconsistency is exasperated when considering the resulting semantics of assignments such as

```java
i = j;
x = y;
```

Here, `i` takes `j`’s value, but `i` and `j` are still distinct in memory. However, `x` and `y` become references to the same widget instance. These two lines use the same syntax, but have completely different meanings.

### 2.3 Suitable environment

The Python interpreter is an excellent pedagogical tool. Though it can be used to execute commands from a source code, it can also be used as a purely interactive environment. Students can begin with such interactive sessions as their first experiences in front of the computer. Each command is evaluated as it is entered, allowing the user to observe the effects of the command or to be informed of any error which occurred. For example, the following Python sessions shows a sample interaction with an instance of the list class.

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

The later transition from using purely interactive sessions to writing source code in separate files is rather straightforward. It is almost precisely as if the keystrokes saved in the file were typed directly into an interpreter session. The only additional issue is the introduction of print and **raw_input** for controlling user interactions.

Notice that this use of the interpreter is quite similar to the goals of pedagogical IDEs such as DrJava [10] and BlueJ [27]. DrJava provides an interactive session in which Java commands can be evaluated individual. Among its features, BlueJ supports the concept of an “object bench” which allows users to instantiate and manipulate objects outside of the context of a typical program. While those are great tools for getting students started in Java, they are not part of a standard Java distribution and thus limit somewhat the transparency and portability of a curriculum.

The interpreter continues to serve as a valuable tool beyond those early days. Rather than starting an interactive session from scratch, it can be scripted by executing code from a file (for example using the `-i` flag from the command line). This allows a user to further inspect and manipulate the objects defined from within their source code. Python’s interpreter also supports forms of immediate documentation. For example in the above session, the command `type(groceries)` will report the type of object currently associated with the identifier, `groceries`. The command, `dir(groceries)`, reports all accessible members of that object, and `help(groceries)` provides more complete documentation on that object.

Python’s standard distribution also includes a lightweight, graphical IDE known as IDLE.
2.4 High-level
As is the case with Java, Python meets Kölling’s main requirements for this criteria in that all memory management is handled by the system, including garbage collection.

One issue of note which distinguishes Python from Java and C++ is the use of self as an explicit first parameter for methods of a class. In all three languages, an invocation of a method, such as thing.foo(x), causes not just parameter x to be sent by the system but also a reference to the object, thing, upon which the method was invoked.

Python makes the transmittal of that self-reference explicit in the function declaration, using a signature,

```python
def foo(self, val):
```

Furthermore, that identifier must be explicitly used to access any members of the instance. This is a rare situation in which Python has more explicit syntax than in Java or C++.

The distinction between instance variables and local variables is at the core of object-oriented programming. The Python style can provide a clear and consistent syntactical treatment (self.blah vs. blah).

In Java, the distinction is typically blurred. The expression, blah, within the context of a method body might be an instance variable or might be a local variable, depending upon the earlier declarations. In fact if declared in both contexts, the local variable masks the instance variable.

2.5 No redundancy
In general, Python measures very well in this regard. The core syntax of the language is quite simple. There are three primary control structures (if, for, while) with a clear distinction between the sequence iteration of a for loop and the conditional repetition of a while loop.

There is an ever slight redundancy in that there are convenient literal forms for the built-in containers, therefore the expression list( ) is identical in meaning as [ ], as are dict( ) and { }. Also, string literals can be delimited either by single quotes ('hello'), double quotes ("hello"), or triple quotes ('''hello''' or """hello"""). With the latter form allowing multilime literals. This flexibility offers convenience, for example when wanting to enclose one of the delimiting characters within a literal.

2.6 Small
Python compares favorably in this regard, relative to the other languages analyzed by Kölling. The core syntax of the language is quite small, with relative few keywords and little extraneous syntax.

However the complexity of the language grows if you consider the API for the built-in classes and libraries. For example the string class supports 34 methods (not including its operators). While this menu of behaviors is extremely valuable for accomplishing flexible text-processing, its use can be overwhelming to a beginning student.

Moreso, the wide range of supported behaviors for built-in classes creates another form of redundancy. Tasks which could be programmed through standard control functions might also be accomplished with a combination of existing functionality.

2.7 Pure object-orientation
Kölling defines of “pure” in this context as a language that does not support any paradigm other than object-orientation. In fact, an earlier version of the paper [26] states, “this means that every program is written as one or more objects, and that objects are the fundamental construct for building systems.”

By this measure, Python clearly does not qualify as pure. It can readily be used as an imperative language and even as a functional language if desired. Indeed, variables and functions can be declared at the top-level scope and arbitrary commands issued without the need for a class definition.

This is in stark contrast to Java, in which a main routine must be defined statically within the context of a class.

There is actually a certain advantage to this flexibility. Java’s strict view of top-level control is a serious obstacle for beginning programmers. In fact it is becoming increasingly common for pedagogical IDEs to intentionally shield students from writing their first programs with a classic, static main routine. More generally, this touches on the debate of the balance between how object-oriented principles should be introduced. Top-level scripting in Python allows students to instantiate and manipulate objects, thereby exploring the concept of an object through interaction, before being prepared to define their own objects.

More significantly, Python is consistently object-oriented in its design. There is a single object model and essentially all entities are treated as first-class objects. This includes all primitive types, functions, classes and even modules. The prevalence of first-class objects does present a few unintended pitfalls for beginners. Since classes are first-class objects, there is a distinction between the following commands:

```python
guests = list()  # guests identifiers a new instance
guests = list # guests identifiers the class itself
```

In similar spirit, since a function is an object, there is a discrepancy between the function and an invocation of the function:

```python
term = word.lower()  # downcased version of word
term = word.lower # the method itself
```

Another concern for purists is that Python does not provide rigorous access control. Instead it provides latent support for the principle of encapsulation based on naming conventions. Names which begin with an underscore are considered to be public; names which begin with a single underscore are akin to protected and names which begin with (but do not end with) a double-underscore are akin to private.
The “protected” semantics of a single underscore has the following implications. Those variables, functions and classes will not be included from a module when using the wildcard syntax, from foo import *. However an identifier can be explicitly imported as from foo import _hidden. Underscored members of a class do not appear explicitly in the online documentation when using help, however those members can be directly accessed by name, if desired.

The “private” double-underscored names are further hidden by explicit name mangling. This helps avoid name-conflicts when inheriting from a base class. However the mangling technique is predictable, and so a a programmer who is aware could access these members with the mangled name.

Though Kölling would clearly consider this lack of rigorous enforcement a weakness (as was his criticism of Smalltalk), we wish to draw attention to a distinction he makes elsewhere in the paper. He points out “that some issues [...] which are often considered extremely important for production programming languages, are of little significance for a teaching language.” As long as students are taught the principle of encapsulation and to follow the conventions, it seems less significant whether the system truly enforces the control.

2.8 Clean concepts

Kölling states, “the concepts that we want to teach should be represented in the language in a clean, consistent and easy-to-understand way.” This is perhaps the most significant criteria for developing a coherent introductory course. The primary goal is to teach the concepts, not the language, yet the language choice has an indirect effect on the presentation. Kölling warns that “we should also not let the language dictate what we have to talk about in first year lectures.” This is at the heart of the classic (though trite) complaint about public static void main(String[] args) in Java. This single line draws attention to at least six concepts that are not appropriate for beginning students.

Python strikes a very nice balance in supporting object-oriented concepts yet in a way which does not push extraneous syntax to the forefront. This allows for a progression over a semester in which the instructor chooses the emphasis, while still begin consistent with earlier lessons.

Type System. Python is both strongly and dynamically typed. Type awareness is an important principle of object-orientation, and students should be taught that each object has an underlying type which is inherent. Yet the dynamic typing reduces the syntactic overhead when using objects. The command,

interest = 'cooking'

associates the identifier, interest, with the underlying string, 'cooking'. Intuitively the student can recognize this object as a string of characters without being explicitly concerned whether the name of that class happens to be str, string or String.

Attention to type information can be reinforced when desired. For example the expression, type(interest), reports the underlying type of this object. When inheritance is later used, a distinction can be drawn between use of type versus isinstance for type-checking.

The impact of dynamic typing also carries over to the declaration of function and methods. The principle of a function signature is a fundamental concept. Parameters can be sent by the caller and a return value sent back by the callee. The caller and callee must share a common agreement as to the expectations for that communication. However the typing of parameters and return values is not explicit in the declaration. Again we find that this allows us to use the concept as we wish but with less syntactic overhead. So long as everyone does what is expected, things will work. We can later introduce language for explicit type-checking if we wish, but this does not need to be done at the beginning.

Polymorphism. One form of polymorphism supported by Java is through the use of interfaces, which can unify a series of distinct classes that share a common set of method signatures. While this serves as a very nice object-oriented design technique, the user of interfaces in a beginning course adds additional syntactic overhead. The interface itself must be defined, and then the class must explicitly declare that it implements the interface.

This form of polymorphism can be explained much more transparently in Python. Since method invocations are performed at runtime, one can simply assume that the object supports the expected behaviors. If one of the behaviors is not supported, a clear run-time error will be reported when an attempt is made to invoke a missing method.

Inheritance. The use of inheritance is quite natural, with minimal syntactic overhead. The declaration for a class which is being defined from scratch begins with the syntax,

class Student:

If using inheritance, such a class is declared as,

class Student(Person):

Multiple inheritance is supported with a syntax,

class Button(Text, Rectangle):

Documentation. Python has optional support for embedding documentation for classes and functions, directly into source code through Python “docstrings.” Again, this is an important principle of good software design, yet one which can be introduced when desired by the instructor.

Encapsulation. As discussed in Section 2.7, encapsulation is accomplished by convention. When first implementing classes, this allows us to discuss the principle of encapsu-
lation and good class design, yet without an explicit requirement for using keywords such as public, protected and private.

Operator Overloading. Python allows operator overloading for user-defined classes through the use of specially named methods (e.g., `__add__` which supports `+`). Yet the technique can easily be ignored by an instructor without consequence. A more pure method-calling syntax can be defined with the method name of choice (e.g., `add`).

2.9 Safety

Kölling states that “the principle of safety is that errors that can easily be detected by a compiler or the runtime system should be detected. Furthermore, they should be detected early, and clear messages should be given about their cause.” We agree with this sentiment. Yet we disagree with Kölling’s subsequent conclusion that teaching languages should be statically typed. On this point, Kölling explicitly admitted his disagreement with earlier views of Sakkinen [42] which distinguish between languages for “exploratory programming,” typically dynamically typed, versus those for “software engineering,” typically statically typed. Given this choice, Kölling states that “in most modern university courses we aim to prepare students for a software engineering view of the world, thus favouring languages of the second kind.”

We strenuously object to this characterization, particularly in the context of a teaching language for an introductory course. This explanation seems to go against his earlier view that the first-year should concentrate on teaching programming, and not on preparing students for industry languages. There is no better time for exploratory programming than in a first course. Furthermore, the rigid enforcement of compile-time type checking causes an unfortunate hurdle for many students, as they are unable to execute any part of their program until they can get the entire program to compile. This leads to a two-phase challenge for novices, getting the syntax correct and then subsequently the semantics. Sadly, many students get stuck at the first hurdle or get overly confident in the quality of their software when surounding only the first hurdle.

As an interpreted language, with the exception of parsing errors, all other errors appear at runtime. Python errors are handled by throwing exceptions. Uncaught exceptions will cause program termination with a display of the current call stack. This will tell the student exactly where the error occurred and the cause of the error. This aids in program development and debugging, offering a more seamless development cycle without the separate hurdle of a compilation. The most nefarious of errors due to the dynamic nature of Python is that a misspelled identifier on the left-hand side of an assignment may go unnoticed.

As for other safety issues, Python’s built-in sequences perform boundary checking. There are no explicit pointers, and there is no such thing as an uninitialized variable. As with Java, python does not allow `= =` and `==` to be confused in the context of a boolean condition. However Python does allow for non-boolean types to be used in place of a conditional, with semantics that are convenient for an experienced program, yet which may lead to subtle errors. A particular misleading example is the evaluation of the expression, `response == 'y' or 'yes'`, which is always `True` as a condition. See Section 2.7, regarding the pitfalls associated with the first-class nature of functions and classes.

2.10 Support for correctness

Python is on par with Java in this category, as neither support true design-by-contract. Both support the use of assert for run-time checking of assertions.

2.11 Easy transition to other languages

We will discuss the transition from Python to other languages as part of Section 3.

3. Our Experiences in CS1 and Beyond

3.1 Getting Started in CS1

Our goal at the onset of our course is to establish a clear and intuitive object-awareness for our students. We emphasize the concept of an object’s underlying state and its set of supported behaviors, using terminology which is standard for the paradigm. These lessons are reinforced at first by examining and manipulating objects from existing classes, and making use of the Python interpreter as a pedagogical tool (as discussed in Section ??).

A typical challenge when introducing object-orientation is providing students with beginning examples of intuitive, tangible objects with which to interact. Often, this barrier is overcome by having students use of custom classes that are not part of the standard language, such as microworlds or graphics. Graphics can indeed provide wonderful examples, and we too use a custom graphics package as part of our course. However an over-reliance upon custom packages decreases the transparency and portability of a curriculum.

With Python, there is a rich set of built-in classes, providing immediately examples of both mutable and immutable classes with which to experiment (e.g., `str`, `list`, `dict`, `file`). Though large API are common in object-oriented languages, they must often be imported and their usage may be beyond the reach of a beginning programmer. We have been particular fond of using instances of the list class among the first objects with which our students interact (we will show a sample session of such an early use in Section ??). Not only are lists an important container type for later in a course, but they can serve to demonstrate most basic object-oriented principles.

We can instantiate one or more lists and see that each has its own mutable state. The member functions include mutators and accessors and demonstrate a variety of signatures. For example, `append` is a mutator which takes one param-
1 class Point:
2     def __init__(self):
3         self._x = 0
4         self._y = 0
5     def getX(self):
6         return self._x
7     def setX(self, val):
8         self._x = val
9     def getY(self):
10        return self._y
11     def setY(self, val):
12        self._y = val

Figure 1. The implementation of a BasicPoint class.

eter and has no return value, insert demonstrates the use of two parameters, and sort provides an example of a signature without parameters or a return value. The pop method is particularly interesting as a mutator that provides a return value (the removed element) and which accepts an optional parameter specifying the index of the element to remove (by default, the last element). The methods, count and index, serve as typical examples of accessors. Furthermore, lists support a variety of convenient operator syntaxes, such as, `groceries[2]` and `'milk' in groceries.

3.2 Class Definitions

When it comes time to implementing a user-defined class, we begin with a very simple example, such as the BasicPoint defined in Figure 1. The high-level syntactic structure is quite similar to the style seen throughout Python, with an initial declaration followed by an indented body of code. The individual methods are defined using a syntax similar to stand-alone functions.

The two novel concepts to highlight are the use of the constructor for establishing attributes of a new instance, and the role of the implicit parameter, `self`, as a reference to the particular instance upon which a method is invoked. Since Python is dynamically typed, the data members of a class are not explicitly declared as part of the original class definition. Instead they are introduced from within the body of the constructor\(^1\). The behavior of the constructor is based upon a specially named method, `__init__` shown starting at line 2 of Figure 1. Access to members of an instance require an explicit qualifier, in this case, `self`. Line 3 of our example, `self._x = 0`, establishes an attribute of the new point instance with the name, `_x`, and the value 0.

\(^1\) Technically, it is possible to add members in other contexts, but this is not a technique which we introduce to students.

1 from math import sqrt  # needed for computing distances
2 class Point:
3     def __init__(self, initialX=0, initialY=0):
4         self._x = initialX
5         self._y = initialY
6     def scale(self, factor):
7         self._x *= factor
8         self._y *= factor
9     def distance(self, other):
10        dx = self._x - other._x
11        dy = self._y - other._y
12        return sqrt(dx * dx + dy * dy)  # imported from math module
13
14 def normalize(self):
15    mag = self.distance( Point( ) )
16    if mag > 0:
17       self.scale(1/mag)
18
19 def __str__(self):
20    return '<' + str(self._x) + ', ' + str(self._y) + '>'
21
22 def __add__(self, other):
23    return Point(self._x + other._x, self._y + other._y)

Figure 2. A more robust Point class implementation (with setX, setY, getX, setY omitted).

Additional Lessons. Moving beyond this first example, we demonstrate additional techniques in the context of a more interesting Point class, shown in Figure 2. The constructor accepts the use of optional parameters (line 4). The scale method serves as a mutator which has a non-trivial effect on several underlying attributes (in contrast to the standard setX mutator). The distance accessor demonstrates the use of a second instance of the same class as a parameter, and the distinction when accessing members (as with `self._x` vs. `other._x` at line 25). We also make use of the qualified math.sqrt function (which itself was imported at line 1). Our implementation of normalize demonstrate the syntax for invoking other methods of the class from within the body of a method (specifically distance and scale). We also see the syntax, `Point( )`, used at line 30 when instantiating a representation of the origin for comparison. Finally, our implementations of `__str__` and `__add__` demonstrate Python’s mechanism for overloading operators, with the former used to control how our point instance is converted to a string, and the latter used to generate a third point which results from an expression, `a + b`, in this context.
def __mul__(self, scalar):
    return Point(scalar * self.x, scalar * self.y)

def __rmul__(self, scalar):
    return Point(scalar * self.x, scalar * self.y)

def __lshift__(self, angle):
    mag = self.distance(Point( ))
    try:
        ang = _math.atan(self.y / self.x)
    except ZeroDivisionError:
        ang = .5 * _math.pi
    if self.x < 0.:
        ang += _math.pi
    ang += _math.pi * angle / 180.
    return Point(mag * _math.cos(ang), mag * _math.sin(ang))

if __name__ == '__main__':

Figure 3. Simple unit testing for our Point class.

Unit Testing In Java, the static main method for a component class can be used as a unit test which is executed when that class is indicated as the primary for execution, but not when that class is imported from within another file. The standard mechanism for such a unit test in Python is provided by following a class definition with a conditional statement indicated as

    def __mul__(self, scalar):
For example, Figure 3 shows the beginning of a potential unit test as a followup to our Point class from Figure 2. If we placed the Point class definition and this unit test in a single file, say Point.py, the main test will be executed when starting the interpreter with the command python Point.py, but it will not be executed if the command, import Point were issued from some other context.

3.3 Advanced Topics in CS1

Because of the decreased emphasis on syntax, we have found that we are able to cover far greater depth in CS1, including some coverage of a handful of advanced topics, such as recursion, data structures, event-driven programming or network programming. Just as the fundamental concepts are easier to present in Python, there is great language support for these additional topics.

3.4 Transition to other languages

Kölling states that “concepts learned with the first teaching language must be easily transferable to the next language following it.” Knowing that this transition was around the corner, we make sure that our first Python course teaching core fundamentals that will be directly applicable in other languages. Our students do not need to unlearn anything from the first course. The biggest challenges in the transition to Java or C++ involve teaching the additional overhead of increased syntactic and semantic issues.

For either transition, there will be some superficial adjustments, such as the designation of block structures, the names of types, or the distinction between for loop syntax. One of the more significant lessons will be moving from a truly interpreted to compiled language. Closely related is the issue of static typing. This effects the declaration of local variables, the declaration of instance variables in a class definition, and the typing of all parameters and return values for functions. However this is not really a foreign concept. Our students have been quite aware of data types, even though not explicit in the syntax. They know quite well what they expect a caller to send as a parameter to their function. They know what type they have been using for each data member of a class.

Another major issue involves understanding varying object models. In Java, the only significant issue is the treatment of primitives as value variables; the rest of Java’s model is quite consistent with Python, including the automatic garbage collection. Clearly this is a much higher burden in C++, given the fine-tuned control the programmer has on the storage model, on parameter passing and on memory management. The use of containers and generics is also a legitimate challenge, mostly because it was so simple in Python and somewhat more cumbersome in Java or C++.

4. Conclusion

In this paper, we have highlighted the attractiveness of Python as a teaching language for an object-oriented introduction. This is based both on a formal analysis of the language features as well as our informal experience having used the language in our own curriculum. Python’s greatest strengths are its simpler, consistent and readable syntax, allowing far more time to focus on fundamental concepts and reinforcing techniques, and far less time discussing syntactical idiosyncrasies and inconsistent semantics. Though we recognize the current inertia for the use of Java as the first language, we hope to raise awareness of the advantages of Python.

References


for simplicity and stability in computer science education. In Proc. 35th SIGCSE Technical Symp. on Computer Science Education (SIGCSE) [5], 115–119.


