Teaching an Introductory Computer Science Sequence with Python

Bradley N. Miller
Department of Computer Science
Luther College
Decorah, Iowa 52101
bmiller@luther.edu

David L. Ranum
Department of Computer Science
Luther College
Decorah, Iowa 52101
ranum@luther.edu

Abstract

Learning computer science requires deliberate and incremental exposure to the fundamental ideas of the discipline. This paper will describe our initial experience teaching an introductory computer science sequence using the programming language Python. We will present our position and then use specific examples to show how Python can provide an exceptional environment for teaching computer science.

1 Introduction

We (the authors, David and Brad) like to ski and we are pretty good at it too. But that level of competence only comes with practice, lots and lots of practice. The designers of ski runs help with that practice in that they code the runs by ability level. Green is for beginner, blue for intermediate, and black runs are for those skiers with advanced skills. This does not however suggest that green runs are void of the important elements that skiers need to become better. Beginning runs still require turns, speed control, and the ability to start and stop.

One of the most unsettling things that we observe on the mountain occurs when a beginning skier ventures out on the blue or black runs before they are ready. It can be an extremely frustrating experience watching them as they slowly attempt to pick their way down the hill, often falling, sometimes crying. The bottom line is that if you start out on the black runs, you are likely to get frustrated and give up. You will often fail before you get a chance to succeed

Computer science deals with people who have problems to solve and with algorithms, the solutions to these problems [4]. Computer science was never intended to simply be the study of programming. To be a computer scientist means first and foremost that you are a problem solver, capable of constructing algorithms either from scratch or by applying patterns from past experience.

Learning computer science is not unlike learning to ski in that the only way to be successful is through deliberate and incremental exposure to the fundamental ideas of the discipline. A beginning computer scientist needs practice so that there is a thorough understanding before continuing on to the more complex parts of the curriculum. In addition, a beginner needs to be given the opportunity to be successful and gain confidence.

As students progress through the introductory computer science sequence, we want them to focus on aspects of problem solving, algorithm development, and algorithm understanding. Unfortunately, many modern programming languages require that students jump into more advanced programming concepts at a point that is too soon in their development. This sets them up for possible failure, not because of the computer science but because of the language vehicle being used.

We believe that Python is an exceptional language for teaching introductory computer science students. We are not alone in this belief [10, 11, 8, 1]. Nor are we alone in our belief that there are many shortcomings to teaching Java [7, 5, 3]. Python has many advantages that can help to minimize the failure scenario noted above. In what follows, we will explore some of these advantages.

2 Introduction to computer science

Our typical computer science student enters the introductory course sequence with very little previous experience. Few have done any programming. In general, they are very good students who have expressed a desire to learn about computer science. Our goals for the first course in computer science are to introduce students to basic problem solving approaches. Programming is presented as a notational means for expressing solutions to these problems.

As students progress from the first course into the second, we begin to focus on classic algorithms and data structures that recur in the solution to many problems. Even though these ideas are more advanced, we still assume that the students are beginners. They may be struggling with some of the ideas and skills from their first course and yet they are ready to further explore the discipline. Many students discover at this point that there is much more to computer science than just writing programs. Data structures and algorithms can be studied and understood at a level that is independent of writing code.

Our goals for the second course are similar to the first. We want to continue the exposure to algorithms, data structures, and problem solving. We want students to understand how to build analysis frameworks to compare and contrast solutions. Most importantly, instead of learning a new programming language or a new software development technique, we want students to continue to learn computer science.

Finally, although it is not the focus of this paper it should be noted that we utilize Java in our third semester introductory course. We believe that Java is an excellent programming language for industrial strength software development and in this course we emphasize many modern programming practices that students need to understand. The topics include graphical user interfaces, model/view/controller architecture, threading, XML, Java Interfaces, version control systems, tools, and documentation. At this point in their study, students are prepared to learn these complex ideas.

2.1 Python Support

As we noted above, deliberate, incremental exposure to computer science provides a good framework for students who are learning difficult concepts for the first time. It is a good teaching method if the tools are in place to support it. Python provides this support.

We begin with a very imperative style, focusing our attention on the basic ideas of algorithm construction. The students quickly gain comfort and confidence with basic programming constructs such as iteration and selection. The ability to provide many small projects with no additional overhead means that there is time for additional practice. This can quickly

move to writing function definitions which will be identical in syntax and use to the method definitions to come later.

In Python, the first program is simply an interactive session:

```
>>> print "Hello Python World"
Hello Python World
```

It is obvious to even the most beginning student what has happened. A single, imperative, algorithmic step, printing a sequence of characters. The result is exactly what you would expect. This sets the stage for the entire introductory sequence. We never want to present a notation without being in a position to understand how and why it is being used.

The use of a language such as Java requires that students be exposed to a program such as:

```
public class Hello{
   public static void main(String[] args){
      System.out.println("Hello Java World");}
}
```

at the start of their study. Unfortunately, a number of the ideas represented by the syntax shown cannot be understood until well into their study. Our goal is to remove the language as a possible bottleneck in the learning process.

In addition to this quick startup, Python provides a consistency that allows a natural transition to the object oriented paradigm. Since everything in Python is an object, the concept of reference is presented on the first day. The assignment statement, $x = 5$, means that x is a reference to the object 5 in the same way that any other piece of data ever used will refer to its value. Students never need to change their basic conceptual model.

The value of this consistency cannot be overlooked as we strive to present difficult computer science. Since Python is inherently dynamic, polymorphic behavior is simply the norm. It is not a new or strange concept and it does not require any special syntactic elements. Instead it is simply the way that an object ought to behave. This simplicity allows students, as they begin to write their own classes, to focus much more on the design of the data and the relationships that exist in the problem being solved.

The interactive nature of the Python environment as well as the ample supply of easy to use programming development tools provide an additional benefit for teaching and learning in the introductory sequence. The Python shell has an interactive prompt where any Python construct, no matter how complex, can be evaluated. This also means that program testing and debugging can be done directly in the shell environment. Variables can be inspected by simply evaluating them. This again allows a very incremental approach to program development. Students can easily know what works and move on to the next, more complex part of the problem.

As another example, consider the typical second program, one that converts temperatures from Fahrenheit to centigrade. In Python, we can simply create a small program as shown below and then invoke it interactively from the Python shell prompt.

```
def main():
    degreesf = input("Enter the temperature in fahrenheit ")
    degreesc = (degreesf - 32) * 5.0/9.0
    print "The equivalent temperature is",degreesc,"degrees C"

>>> main()
Please enter a temperature in fahrenheit 98.6
The equivalent temperature is 37.0 degrees C
```

In Java, we are immediately confronted with the issues of I/O classes and exceptions. The language bottleneck with additional syntactic items that cannot be immediately understood diverts the student's attention from the main task. We should note that there are solutions to this particular Java problem [6].

Listing 1: Java version of temperature converter

```
import java.io.*;

public class Converter {
   public static void main (String[] args) {
      try {
         InputStreamReader isr = new InputStreamReader(System.in);
         BufferedReader br = new BufferedReader(isr);

         System.out.print("Enter the temperature in fahrenheit ");
         double degreesf = Double.parseDouble(br.readLine());
         double degreesc = (degreesf - 32) * 5.0/9.0
         System.out.println("The equivalent temperature is "
                            + degreesc + " degrees C"
      }
      catch (IOException e) {  //handle exception here
      }
   }
}
```

3 Case Studies

In a second course in computer science there are three broad areas in which we want the students to gain experience. In decreasing order of importance these three areas are:

1. Students should get an intuitive feel for abstract data types by working with them directly.

2. Students should get a conceptual understanding of a wide range of algorithms.

3. Students should begin to get accustomed to important programming language features so that they can easily move into another programming language.

In this section we will look at some common programs written in an introductory data structures course. We will compare the programs written in Python, Java, and Pseudocode. As we look at the various example programs we will discuss how each either helps or hinders the three goals outlined above.

3.1 Insertion Sort

We will begin with the simple insertion sort algorithm. Shown below is the pseudocode from [2].

```
1: Insertion-Sort(A)
2: for j ← 2 to length[A] do
3:     key ← A[j]
4:     i ← j − 1
5:     while i > 0 and A[i] > key do
6:         A[i + 1] ← A[i]
7:         i ← i − 1
8:     A[i + 1] ← key
```

In listing 2 we show the Python code for insertion sort. Notice that it is the same number of statements as the pseudocode. The few differences in the pseudocode and the Python code are because the pseudocode assumes that arrays are indexed starting at one, whereas Python indexes lists beginning with zero. Like the pseudocode Python uses indentation to denote block structure in a program. While experienced programmers who are used to the curly braces of C, C++, and Java may find this a step backward, novice programmers find indentation to be quite natural and we have found that it enforces a more readable coding style than students typically adopt when they can use braces.

Because Python is dynamically typed there is no need for variable declarations. The code, as written, will work for many different Python data types. The following Python session illustrates loading the insertion sort code from a file and trying it out on a list of integers, strings, and floats.

```
cray:MICS2005> python -i insertion.py
>>> A = [9, 4, 2, 10, 7, 22, 1]
>>> insertionSort(A)
```

Listing 2: Insertion Sort in Python

```
def insertionSort(A):
    for j in range(1,len(A)):
        key = A[j]
        i = j
        while i>0 and A[i-1]>key:
            A[i]=A[i-1]
            i = i-1
        A[i]=key
```

```
>>> A
[1, 2, 4, 7, 9, 10, 22]
>>> B = ['dog', 'cat', 'ape', 'eel']
>>> insertionSort(B)
>>> B
['ape', 'cat', 'dog', 'eel']
>>> C = [3.14, 2.78, 9.80, 3.8]
>>> insertionSort(C)
>>> C
[2.78, 3.14, 3.8, 9.8]
>>>
```

In listing 3 we show the code required to create a complete Java program for insertion sort. This code is taken from [9], a popular data structures and algorithms book for Java.

Listing 3: Java version of Insertion Sort

```
public final class Sort {
    public static void insertionSort( Comparable [ ] a ) {
        int j;
        for( int p = 1; p < a.length; p++ ) {
            Comparable tmp = a[ p ];
            for( j = p; j > 0 && tmp.compareTo(a[j - 1]) < 0; j--)
                a[ j ] = a[ j - 1 ];
            a[ j ] = tmp;
        }
    }
    public static void main( String [ ] args ) {
        Integer [ ] a = new Integer[ NUM_ITEMS ];
        for( int i = 0; i < a.length; i++ )
            a[ i ] = new Integer( i );
        for( theSeed = 0; theSeed < 20; theSeed++ ) {
            Random.permute( a );
            Sort.insertionSort( a );
        }
    }
}
```

Notice that the basic elements of insertion sort are still quite similar. However, there is so much more going on that it is difficult for a novice programmer to know what to focus on. The fact that the algorithm must be declared `public static void` inside a class adds the complexity of creating a new class just to contain this algorithm. The braces and variable declarations add length to the program. A simple comparison operator must be replaced with a method call. Finally, to make the algorithm sort multiple data types we must employ a Java Interface called Comparable. Interfaces, although extremely powerful, are also extremely complex. So what is a beginning computer science student going to focus on first?

Much of the complexity in the Java example can be traced back to the use of the Comparable interface. The Comparable interface is specified as follows:

Listing 4: Comparable Interface

```
public interface Comparable {
    public int compareTo(Object o)
}
```

This means that any Java class that implements the Comparable interface must implement a `compareTo` method. The Java documentation tells us exactly what `compareTo` should do: *Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.* The built in classes in Java all implement this interface so you do not need to explicitly implement `compareTo` if you are sorting arrays of Strings, Floats, Doubles, or Integers. Unfortunately, if that is the case then you cannot use the method in listing 3 to sort arrays of int, float, or double.

As stated earlier, in Python all variables refer to objects so we do not have the distinction between Integer and int. How does Python handle sorting arbitrary objects? Suppose that we want to use our Python insertion sort function to sort a list of students according to their GPA. We must first define a class to represent each student.

The simplest class definition for a student is shown in listing 5.

Listing 5: Class Definition for Student

```
class student:
    def __init__(self,lname,fname,gpa):
        self.fname = fname
        self.lname = lname
        self.gpa = gpa
```

Once we have defined this class we can test it by interactively creating some instances. We can even put these instances into a list and try to sort them.

```
>>> b = student('miller','brad',4.0)
>>> d = student('ranum', 'david', 3.8)
>>> g = student('gates', 'bill', 3.2)
>>> A = [d,g,b]
>>> A
[<student.student instance at 0x70080>,
 <student.student instance at 0x70418>,
 <student.student instance at 0x67170>]
>>> insertionSort(A)
>>> A
[<student.student instance at 0x67170>,
 <student.student instance at 0x70080>,
 <student.student instance at 0x70418>]
```

Notice that even with this basic class definition, Python provides us with a default behavior that does not create an error. In listing 6 we extend the definition of Student with two additional methods. The __repr__ method and the __cmp__ method provide the student class with a friendly string representation and a comparison. Java programmers should think of these two methods as tostring and compareTo.

Listing 6: A student class that can be sorted by GPA

```
class student:
    def __init__(self,lname,fname,gpa):
        self.fname = fname
        self.lname = lname
        self.gpa = gpa

    def __repr__(self):
        return(self.fname + " " + self.lname + ": "
               + str(self.gpa)

    def __cmp__(self,other):
        if self.gpa < other.gpa:
            return -1
        elif self.gpa == other.gpa:
            return 0
        else:
            return 1
```

The following Python session demonstrates how easy it is to create some class instances, add them to a list and then sort the list. Notice that in this session we now get a much more helpful view of the instances, because python is using the __repr__ method to display them.

```
>>> b = student('miller','brad',4.0)
>>> d = student('ranum','david',3.8)
>>> g = student('gates','bill',2.9)
>>> A = [b,d,g]
>>> A
[brad miller: 4.0, david ranum: 3.8, bill gates: 2.9]
>>> insertionSort(A)
>> A
[bill gates: 2.9, david ranum: 3.8, brad miller: 4.0]
```

3.2 Trees

Having examined a simple algorithm, we will turn our attention to a data structure. A common introductory data structure is the Tree. In Many algorithms textbooks use a recursive definition of a tree as follows:

> Either a tree is empty, or it consists of a root and zero or more subtrees, each of which is also a tree.

Unfortunately implementing the recursive definition is very difficult in Java or C++. Most data structures textbooks define an outer class called tree, with an inner class called TreeNode that maintains the relationships between a node and its children.

In Python there is a nice progression that we can follow from a very simple recursive tree data structure using lists to a recursive class based data structure. This progression allows the students to get comfortable with the concept of a tree, and throughout the progression emphasizes the importance of abstracting the representation of a data structure away from its operations.

To get started with trees and get a basic understanding of the data structure we can use a very simple list of lists representation. In a list of lists representation for a tree we will store the value of the root node as the first element of the list, the second element of the list will itself be a list the represents the left subtree, and the third element of the list will be another list that represents the right subtree. To illustrate this storage technique consider the following example. Figure 1 shows a simple tree. The corresponding list implementation for this tree is:

```
>>>  myTree = ['a',
...             ['b', ['d', [], []], ['e', [], []] ],
...             ['c', ['f', [], []], []] ]
```

Notice that we can access subtrees of the list using standard list indices. The root of the tree is myTree[0], the left subtree of the root is myTree[1], and the right subtree is myTree[2].

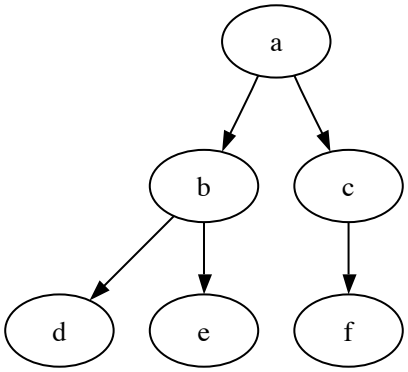

Figure 1: A small example tree

The following Python transcript illustrates creating a simple tree using a list. Once the tree is constructed we can access the root, left, and right subtrees.

One very nice property of this list of lists approach is that the structure of a list representing a subtree adheres to the structure defined for a tree. A subtree that has a root element and two empty subtrees is a leaf node Another nice feature of the list of lists approach is that it generalizes to a tree that has many subtrees. In the case where the tree is more than a binary tree subtree is just another sublist.

```
>>> myTree
['a', ['b', ['d', [], []], ['e', [], []]],
      ['c', ['f', [], []], []]]
>>> myTree[1]
['b', ['d', [], []], ['e', [], []]]
>>> myTree[0]
'a'
>>> myTree[2]
['c', ['f', [], []], []]
```

With this representation of a tree we can now write some simple functions to access the parts of the tree. At this point it is a very natural progression to begin talking about operations on an abstract data type called a Tree. In listing 7 we show a few simple access functions and the code for an inorder traversal.

The following interactive Python session demonstrates the use of the functions defined above.

```
>>> T
[7, [5, [3, [], []], [6, [], []]], [11, [8, [], []], []]]
>>> getRootVal(T)
7
>>> getLeftChild(T)
[5, [3, [], []], [6, [], []]]
```

Listing 7: Python code for simple tree operations

```
def getRootVal(root):
    return root[0]

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]

def inorder(tree):
    if tree != []:
        inorder(getLeftChild(tree))
        print getRootVal(tree)
        inorder(getRightChild(tree))
```

```
>>> getRightChild(T)
[11, [8, [], []], []]
>>> inorder(T)
3
5
6
7
8
11
```

In part one of our tree discussion we have introduced some functions to operate on a tree and access parts of a tree where the structure is very apparent. In part two of the discussion can move on to a class implementation that represents a tree as an abstract data type. At this point it even makes sense to talk about a different internal representation for the tree. In listing 8 we show part of the class definition for a binary tree that directly implements the recursive definition above.

If you look at the listing carefully you will see that when we call getRightChild we are in fact returning a BinaryTree object. At this point we may choose to implement the tree traversal as a method of the abstract data type, or we may implement it as an external function using the methods of the tree as before. In fact now is a good time to show our students that even though we have completely changed the implementation of our data structure we only need to make one tiny change to the external `inorder` function in order for it to work with this new tree.

In the following example we construct and traverse a tree built with our new implementation of inorder. For students who may still be struggling with the use of `self` in Python's object system, this sequence provides an excellent time to reinforce the use of self as an implicit parameter. You can simply point out that the tree parameter passed to the various

Listing 8: A recursive binary tree abstract data type

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
        self.left = None
        self.right = None

    def insertLeft(self,newNode):
        if self.left == None:
            self.left = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.left = self.left
            self.left = t

    def getLeftChild(self):
        return self.left
```

Listing 9: Inorder that works with new tree abstract data type

```
def inorder(tree):
    if tree != None:
        inorder(tree.getLeftChild())
        print tree.getRootVal()
        inorder(tree.getRightChild())
```

functions has been replaced by the instance of tree to the left of the dot. However a tree is still passed implicitly to the function through the parameter called self.

```
>>> t = BinaryTree(7)
>>> t.insertLeft(3)
>>> t.insertRight(9)
>>> l = t.getLeftChild()
>>> l.insertLeft(1)
>>> l.insertRight(4)
>>> inorder(t)
1
3
4
7
9
```

Although we have only shown you a few short examples of Python programming we hope that you can see that it has great advantages for cleanly programming important algorithms. In addition we have tried to demonstrate a few key language features such as polymorphism and operator overloading. We believe that the language features along with the kind of exploratory programming that Python encourages will help students be well prepared for the next stage of their educational development in Computer Science.

4 Conclusion: Python Works

Teaching and learning computer science using Python has been and continues to be a positive experience. Students are more successful and have gained and shown more confidence. In a single semester our students completed 27 programming exercises. This was over twice that of previous years. In addition, more students successfully completed all projects.

From a teaching perspective, Python is very gratifying. It has a clean, simple syntax and an intuitive user environment. The basic collections are very powerful and yet easy to use. The interactive nature of the language creates an obvious place to test data structure components without the need for additional coding of driver functions. Finally, Python provides a textbook-like notation for representing algorithms alleviating the need for an additional layer of pseudocode. This allows the illustration of many relevant, modern, and interesting problems that make use of the algorithm and data structure ideas.

Although it may appear that teaching Java as an introductory language is a foregone conclusion, we believe that it does not need to be the case. We believe that it is advantageous for beginning students to spend time on the green runs learning the rudimentary ideas relating to algorithms and data structures. If you are among those of our colleagues who are frustrated with Java or C++, we encourage you to consider Python and join us on the green runs, it works.

References

[1] P. H. Chou, *Algorithm education in python*, Proceedings of Python 10, 2002.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, MIT Press, 2001.

[3] ACM Java Task Force, *Taxonomy of problems in teaching java*.

[4] A. I. Forsyth, T. A. Keenan, E. I. Organick, and W. Stenberg, *Computer science: A first course*, John Wiley, 1975.

[5] Jeremy D. Frens, *Taming the tiger: teaching the next version of java*, SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education, ACM Press, 2004, pp. 151–155.

[6] Viera K. Proulx and Richard Rasala, *Java io and testing made simple*, SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education, ACM Press, 2004, pp. 161–165.

[7] Eric Roberts, *The dream of a common language: the search for simplicity and stability in computer science education*, SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education, ACM Press, 2004, pp. 115–119.

[8] Frank Stajano, *Python in education: Raising a generation of native speakers*, Proceedings of Python 8, the International Python Conference, 2000.

[9] Mark Allen Weiss, *Data structures and problem solving using java*, Addison Wesley, 2002.

[10] John M. Zelle, *Python as a first language*, Proceedings of 13th Annual Midwest Computer Conference, 1999.

[11] _____, *Python programming: An introduction to computer science*, Franklin Beedle, 2003.