AC 2009-1781: USE OF PYTHON IN TEACHING DISCRETE MATHEMATICS

Ali Farahani, National University, San Diego

USE OF PYTHON IN TEACHING DISCRETE MATHEMATICS

Alireza Farahani, Ronald P. Uhlig
School of Engineering and Technology, National University,
11255 North Torrey Pines Road, La Jolla, California, U.S.A.

Abstract:

Discrete structures class is among the foundational classes in computer science since it includes materials that are highly pervasive in other areas of computer science such as data structures and algorithm design. A course in discrete structures typically starts with an introduction to logic and mathematical proof techniques then the focus is placed on graph theory with algorithm design and analysis techniques. These topics tend to be difficult to teach and hard to grasp students since materials are often abstract and at times disconnected. In this study we examine the impact of using Python programming language as an aid in teaching a discrete structures class to computer science students. Python is a programming language that supports both object–oriented and functional programming paradigms. It is easy to read and understand with syntax suitable for algorithm development. Python can run in an interactive mode which provides an efficient environment for students to experiment with abstract ideas in order to better understand the concepts.

Introduction:

A discrete structures course is considered to be a foundational in computer science. This is where most students are exposed to formal rigorous mathematical treatment of theoretical aspect of computer science. By the time most students reach this course they have already developed an introductory to intermediate level programming skills. Few educators have attempted to integrate programming into teaching of discrete mathematics. P Chou[1] reported using Python in algorithm class with overall positive student evaluation result, however cautioned that improvements must be made.
This author happened to believe integrating programming into the discrete structures class can enhance student learning through experimentation. In particular, Python language is most suitable for this purpose because of its intuitive syntax and semantics and close similarity with mathematics notation and algorithmic psudocode. Python provide an interactive window for code testing and simple calculations. This paper described some of the ways in which Python can be employed in teaching discrete structures

System Requirements:

Python is a general purpose programming that is both elegant and pragmatic with a simple intuitive syntax and semantics. The language is very powerful yet it is simple; it is used by both novice and export developers. Python runs on all major hardware platforms and operating

systems. One can quickly become productive with python. It has rich standard library and many other modules. The language is easy to learn so it is suitable for new programmers as well.
The codes in this paper were test using Python 2.6 with windows XP operating system. Python is freely available for downloading and the installation is seamless.

Discrete structures course:

All Computer Science students take discrete structures class as a required course. It is in this course that students start to look at things more formally and rigorously, the rigorous nature of the course and the coverage of mostly mathematical concepts tend to be challenging and at times hinder understanding of the some of the concepts. For computer science student a more hands on and programmatic approach to teaching of some of the concepts in discrete structures may be more suitable. Python has a simple intuitive syntax that student can easily grasp and adapt to in a very short time and start experimenting with the concepts. A typical computer science course in discrete structures starts with sets and logic along with some combinatorial counting techniques, it then moves on to mathematical proofs which is followed by an exposure on algorithm design and analysis. The following sections demonstrate some examples where Python can be utilized to support teaching of some topics in discrete mathematics:

Set operations:
The concept of sets and operations on sets are fundamental in discrete mathematics; Python has a powerful built in *list* type and *set* object that can easily be used to experiment with construction of sets as well as operations on them. A *list* type in Python can be a heterogeneous collection which can be modified. Often in a discrete mathematics course a set builder notation is used to construct a set. For example, the set of the first twenty even numbers using set builder notation is denoted by $S = \{x \mid x = 2n; 0 \leq n \leq 19\}$. In Python this set can easily be specified by

```
S = [2*x for x in range(19)]
```

The syntax is very intuitive and maps well to its counterpart in mathematics. Once a set a built, it is easy to index though its elements in a simple loop. The next code snippet reports on whether N is a prime number; it simply applies the definition of a prime number to N by looping through a set of integers less that N to see is N has a factor there.

```
for i in [x+1 for x in range(N)]:
          if N % i==0 and (i!=1 and i!=N):
               return False
     return True
```

Once again the syntax and semantics are very intuitive and readily graspable by a computer science student. In Python set membership can be examined by the command "in". Students can easily experiment with introductory set theory concepts using Python's set object. Here is an example of set equality; suppose we want to verify that the sets A and B given below are equal.

$$A = \{x \mid x^2 + x - 6 = 0\} \text{ and } B = \{2, -3\} \text{ then } A = B$$

In Python set A can be constructed as

```
A=set ([x for x in range (-50,50) if x**2+x-6==0])
```

where the *set* command is applied to convert the *list* to a *set* object. The set is constructed by searching for integer solutions of the quadratic equation in a specified range. Set B is simple to construct in Python, B`=set ([2,-3]).` Now we use the command `A==B` to verify the equality. The system returns "True".
A set product, or a set of ordered pair is easily built by a single line of code assuming that sets A and B have been defined then `[(a,b) for a in A for b in B]` produces the product of the two sets. The *set* union, intersection and difference operations are all available in Python thus enabling one to verify Demorgan's law for sets.

Logic:
In logic students typically are requires to generate truth tables for a variety of Boolean expressions. Python directly supports conjunction and disjunction in terms of the build in primitives *and* and *or,* the negation is done using *not*. A nested loop can be constructed to generate the truth table for standard logic operators such as the *and*

```
for p in (True, False):
        for q in (True, False):
             print "%10s %10s %10s" % ( p, q, p and q)
```

Once again the code is highly intuitive and requires minimal coding ability. The print statement is formatted to tabulate the output. Students can use Python to instantly verify their logic work, consider the code segment that generates the truth table for the expression $\neg p \wedge (q \vee r)$ which is easy to understand and can help support student's grasp of the concept.

```
for p in (True, False):
  for q in (True, False):
       for r in (True, False):
          print "%10s %10s %10s %10s" % ( p, q, r, not p and (q or r))
```

The implication statement $p -> q$ can either be looked at as equivalent to $\neg p \vee q$ or encoded as a function

```
def implies(p,q):
        if a:
                return b
        else:
                return True
```

To express the logical statement $x \geq 0$ and $y \geq 0 \rightarrow x * y \geq 0$ we can write *implies ($x \geq 0$ and $y \geq 0$, $x * y \geq 0$)*

Universal and existential quantifiers:
A propositional or a predicate function $f(d)$ is a Boolean function whose truth value depends on the value of d which is restricted to a properly defined domain. Most of the statements in mathematics and computer science use terms such as "for every ($\forall$)" or "there exists($\exists$)", as an example consider the following propositions $\forall x(x^2 \geq 0)$ or $\exists x(x-2=0)$. The symbols $\forall$ and $\exists$ are called universal and existential quantifier respectively. Once again the nested loops can aid us in verifying the truth value of the quantified statements. To verify $\forall x \in Z(x^2 \geq 0)$

```
for x  in range(-100,100]:
            if not (x**2 >=0):
                    return False
      return True
```

The code only test integer numbers between -100 and 100; if any of the values assigned to the loop variable fails the stated proposition i.e. $x^2 \geq 0$, then the for loop is terminated with a "False" output, otherwise the for loop will iterated exhaustively and outputs True. Similarly the existential quantifier $\exists x(x-2=0)$ can be tested by the code

```
for x  in range(-100,100)]:
            if x - 2 == 0:
                    return True
      return False
```

This can easily be extended to nested quantifiers as well. For example, $\forall x \exists y(x-y=0)$ can be tested

```
for x in range(-100,100)]:
   for y in range(-100,100)]:
            if i - j == 0:
                    return True
      return False
```

Combinatorial selection:
Furthermore, the combinatorial problems can easily be formulated in terms of loops. Here is an example to print all possible ways to select 3 out of the N first non-negative integers if order is relevant and repetition is allowed.

```
for i1 in range(0,3):
      for i2 in range(0,3):
            for i3 in range (0, 3):
                  print i1,i2,i3
```

By simple adjustments of the range in the inner loops one can generate a selection without repetition or a larger selection can be obtained with an appropriate number of nested loops.

Algorithm:
Algorithm design is essential to computer science; algorithms have been devised to solve many problems. Most introductory algorithm design and analysis courses start with either searching or sorting algorithms. Once again Python's intuitive syntax, interactive nature and close similarity to pseudocode describing the algorithm help in teaching the concepts. As an example consider the card shuffle algorithm in the function below. The code is easily understood and can be tested immediately in an interactive mode. Notice the elegant way for swap in the last line of code. What is interesting is that the shuffle code works on any type of list.

```
def shuffel(n):
        for i in range(1,len(n)):
                j=random.randint(i,len(n)-1)
                n[i],n[j]=n[j],n[i]
```

Bubble sort is another good example to show the elegance and ease with which seemingly complicated algorithms can be coded and tested

```
def bubble_sort ( list ) :
  for i in range ( 0, len ( list ) - 1 ):
    for j in range ( 0, len ( list ) - i - 1 ):
      if list[j] > list[j + 1] :
        list[j], list[j + 1] = list[j + 1], list[j]
```

A discrete structure class is the first course in which student are exposed to graph algorithms. In Python the dictionary type can be used to implement the adjacency list representation of the graph.

Conclusion:

This paper presented ways to integrate Python programming language into teaching of discrete mathematics for Computer Science students. Python enables students to readily and instantly experiment with many of the logic, combinatorial and algorithms concepts typically covered in a discrete structures course. Ability to experiment to better understand the theory can further improve student's problem solving ability as well.

Reference:

1. Pai H. Chou, "*Algorithm Education in Python*" Department of Electrical and Computer Engineering, University of California, Irvine. http://www.ece.uci.edu/~chou/py02/python.html