Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Sandeep Sadanandan

Sommer Semester 2009
Solution Sheet 1
May 16, 2009

Python For Fine Programmers

Problem 1 (6 Points)

Write a set of classes in Python for chess coins. Arrange them so that a base class consists of all the basic details (eg. Color of coin and the Color of square where it is). And the subclasses could inherit from it - all the basic properties and then the subclasses implement the special methods for them. (Queen, Bishop, Knight, Rook, King and Pawn)

Solution

```
def is_valid_pos(pos):
  col = ord(pos[0])
  row = ord(pos[1])
  if col < ord('a') or col > ord('h'):
    return False
  if row < ord('1') or row > ord('8'):
    return False
  return True

columnvariables = ['a','b','c','d','e','f','g','h',]
rowvariables    = ['1','2','3','4','5','6','7','8',]

class ChessCoin(object):

  def __init__(self, type, color, position):
    self.type     = type
    self.color    = color
    if not is_valid_pos(position):
      print "Wrong position to start with"
      return None
    self.position = position
    self.update_movableto()

  def still_in_game(self):
    return self.position != "00"

  def print_position(self):
    print self.position

  def __str__(self):
    return ' '.join(['I am a', self.color, self.type, 'at',
        self.position])

  def cut_me(self):
        self.position = "00"
        self.movableto = []

    def movable_postions(self):
        return self.movableto

    def update_movableto(self):
        pass


class Rook(ChessCoin):

    def update_movableto(self):
        if not self.still_in_game():
            return []
        pos = self.position
        row = pos[1]
        col = pos[0]
        wholecol = [col + p for p in rowvariables if p != row]
        wholerow = [p + row for p in columnvariables if p != col]
        self.movableto = wholerow + wholecol

    def move(self, position):
        if not self.still_in_game():
            return
        if position in self.movable_postions():
            self.position = position
            self.update_movableto()
        else:
            print "Sorry, Cannot move to there"

R = Rook("rook", "black", "a1")

print R.movable_postions()
R.move(’a6’)
print R
print R.movable_postions()

R.cut_me()
print R
print R.movable_postions()
```

Problem 2 (8 Points)

Write a class for Rational numbers. Implement any 3 of the methods which would permit one to use the mathematical operators ($+, *, /, \ldots$ or similar ones)

Solution

```
from fractions import gcd
class myRational(object):
  def __init__(self, p, q):
    if q == 0:
      print "No such RAT is possible"
      return None
    self.numer = p
    self.denom = q

  def __repr__ (self):
    return "I am rational " + repr(self.numer) +
       "/" + repr(self.denom)

  def __str__ (self):
    return repr((self.numer, self.denom))


  def __add__ (self, ano):
    newden = self.denom * ano.denom / gcd(self.denom, ano.denom)
    newnum = ((self.numer * newden / self.denom) + (ano.numer *
    newden / ano.denom))
    return myRational(newnum, newden)

  def __mul__ (self, ano):

    newnum = self.numer * ano.numer
    newden = self.denom * ano.denom

    g      = gcd(newnum, newden)
    return myRational(newnum/g, newden/g)

  def reciprocal(self):
    return myRational(self.denom, self.numer)

  def __div__ (self, ano):
    return self * ano.reciprocal()

x = myRational(12, 34)
y = myRational(22, 34)

print x, y
z = x + y
w = x * y
u = x/y
print z, w, u
```

Problem 3 (8 Points)

Flattening a list.
Write a program, which accepts a list (with sublists, and subsublists) as an input and outputs a single list which has the members of the sublists/subsublists as elements. And input of [1, [2, 3], [[4, [5], 6], 7], 8] would give an output of [1, 2, 3, 4, 5, 6, 7, 8].

Solution

```
def flatten(e):
    if not e:
        return []
    if not isinstance(e, list):
        return [e]
    return flatten(e[0]) + flatten(e[1:])

l = [1, 2, 4]
k = [6, 7, 9]

lk = [l, k]
lkkl = [l, [k, k], l]

print lk
print flatten(lk)

print lkkl
print flatten(lkkl)
```

Problem 4 (4 Points)

Write a class for a node of a binary tree. (Not necessarily a binary search tree). Implement the basic methods.

Solution

```
class Tree(object):
  def __init__ (self, data):
    self.data   = data
    self.lchild = None
    self.rchild = None
    self.below  = 0


  def insert(self, data):
    self.below += 1
    if self.data < data:
      if self.lchild:
        self.lchild.insert(data)
      else:
        self.lchild = Tree(data)
    elif self.data > data:
      if self.rchild:
        self.rchild.insert(data)
      else:
        self.rchild = Tree(data)
    else:
      self.below -= 1
      pass


    def inorder(self):
        if self.lchild:
            self.lchild.inorder()
        print repr(self.data)
        if self.rchild:
            self.rchild.inorder()

    def __repr__(self):

        return repr(self.lchild) + repr(self.data) + repr(self.rchild)

T = Tree(12)
from random import randint

rands = [randint(12, 99) for i in range(1, 13)]
print rands

for rand in rands:
    T.insert(rand)

T.inorder()
```

Problem 5 (5 Points)

Bonus Question.

- Prove that a knight starting to move from a square of a chess board would reach a square of the same color, if and only if it has made even number of moves
- Devise a method to enable a knight, to go through every square of a chess board exactly once in 64 moves.

Solution

- Every jump changes color
- Jump to the position from where there is a least number of possibilities to jump.