```
from __future__ import division
import sys
import random
import math
import numpy as np
from models import *
from options import *
from utilities import *
sys.dont_write_bytecode = True

#say = Utilities().say

class MaxWalkSat():
  model = None
  minR=0
  maxR=0
  random.seed(40)
  def __init__(self,modelName):
    self.model=modelName


  def evaluate(self):
    model = self.model
    #print "Model used: %s"%model.info()
    minR=model.minR
    maxR=model.maxR
    maxTries=int(myoptions['MaxWalkSat']['maxTries'])
    maxChanges=int(myoptions['MaxWalkSat']['maxChanges'])
    n=model.n
    threshold=float(myoptions['MaxWalkSat']['threshold'])
    probLocalSearch=float(myoptions['MaxWalkSat']['probLocalSearch'])
    bestScore=100
    bestSolution=[]


    print "Value of p: %f"%probLocalSearch
   # model = Fonseca()
    model.baseline(minR,maxR)
    print model.maxVal,model.minVal

    for i in range(0,maxTries): #Outer Loop
      solution=[]
      for x in range(0,n):
        solution.append(minR + random.random()*(maxR-minR))
      #print "Solution: ",
      #print solution
      for j in range(0,maxChanges):      #Inner Loop
         score = model.evaluate(solution)
         #print score
         # optional-start
         if(score < bestScore):
           bestScore=score
           bestSolution=solution

         # optional-end
         if(score < threshold):
           print "threshold reached|Tries: %d|Changes: %d"%(i,j)
           return solution,score

         if random.random() > probLocalSearch:
             c = int(0 + (self.model.n-0)*random.random())
             solution[c]=model.neighbour(minR,maxR)
         else:
             tempBestScore=score
             tempBestSolution=solution
             interval = (maxR-minR)/10
             c = int(0 + (self.model.n-0)*random.random())
             for itr in range(0,10):
                solution[c] = minR + (itr*interval)*random.random()
                tempScore = model.evaluate(solution)
                if tempBestScore > tempScore:     # score is correlated to max?
                  tempBestScore=tempScore
                  tempBestSolution=solution
```

```
             solution=tempBestSolution

    return bestSolution,bestScore

def probFunction(old,new,t):
   return math.exp(1 *(old-new)/t)

class SA():
  model = None
  minR=0
  maxR=0
  random.seed(1)
  def __init__(self,modelName):
    self.model=modelName

  def neighbour(self,solution,minR,maxR):
    returnValue = []
    n=len(solution)
    for i in range(0,n):
      tempRand = random.random()
      if tempRand <(1/self.model.n):
        returnValue.append(minR + (maxR - minR)*random.random())
      else:
        returnValue.append(solution[i])
    return returnValue

  def evaluate(self):
    model=self.model
    #print "Model used: %s"%(model.info())
    minR = model.minR
    maxR = model.maxR
    model.baseline(minR,maxR)
    print model.maxVal, model.minVal

    s = [minR + (maxR - minR)*random.random() for z in range(0,model.n)]
    print s
    e = model.evaluate(s)
    emax = int(myoptions['SA']['emax'])
    sb = s                       #Initial Best Solution
    eb = e                       #Initial Best Energy
    k = 1
    kmax = int(myoptions['SA']['kmax'])
    count=0
    while(k ≤ kmax ∧ e > emax):
      sn = self.neighbour(s,minR,maxR)
      en = model.evaluate(sn)
      if(en < eb):
        sb = sn
        eb = en
        print("!"),#we get to somewhere better globally
      tempProb = probFunction(e,en,k/kmax)
      tempRand = random.random()
#      print " tempProb: %f tempRand: %f " %(tempProb,tempRand)
      if(en < e):
        s = sn
        e = en
        print("+"), #we get to somewhere better locally
      elif(tempProb ≤ tempRand):
        jump = True
        s = sn
        e = en
        print("?") #we are jumping to something sub-optimal;
        count+=1
      print("."),
      k += 1
      if(k % 50 ≡ 0):
         print "\n"
       #  print "%f{%d}"%(sb,count),
         count=0
    return sb,eb
```