```
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 SearchersBasic():
  tempList=[]
  def display(self,printChar,score):
    self.tempList.append(score)
    if(self.displayStyle≡"display1"):
      print(printChar),

  def display2(self):
    if(self.displayStyle≡"display2"):
      #print xtile(self.tempList,width=25,show=" %1.6f")
      self.tempList=[]

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



  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(1,maxChanges):      #Inner Loop
         #print "Index : %d"%j
         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)
           self.display(".",score),
```



```
           self.display2()
           self.model.evalBetter()
           revN = model.maxVal-model.minVal
           return bestSolution,bestScore*revN+model.minVal,self.model

         if(random.random() > probLocalSearch):
             c = int((self.model.n)*random.random())
             solution[c]=model.neighbour(minR,maxR)
             self.display("+",score),
         else:
             tempBestScore=score
             tempBestSolution=solution
             interval = (maxR-minR)/10
             c = int(self.model.n*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
             self.display("!",tempBestScore),
         self.display(".",score),

         if(self.model.lives ≡ 1):
           #print "DEATH"
           self.display2()
           self.model.evalBetter()
           revN = model.maxVal-model.minVal
           return bestSolution,bestScore*revN+model.minVal,self.model

         if(j%50≡0):
            #print "here"
            self.display2()
            self.model.evalBetter()
    revN = model.maxVal-model.minVal
    return bestSolution,bestScore*revN+model.minVal,self.model

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

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


  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 "MaxVal: %f MinVal: %f"%(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):
      #print k,e
      sn = self.neighbour(s,minR,maxR)
      en = model.evaluate(sn)
      if(en < eb):
        sb = sn
        eb = en
        self.display(".",en),#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
        self.display("+",en), #we get to somewhere better locally
      elif(tempProb > tempRand):
        jump = True
        s = sn
        e = en
        self.display("?",en), #we are jumping to something sub-optimal;
        count+=1
      self.display(".",en),
      k += 1

      if(self.model.lives ≡ 0):
        self.display2()
        self.model.emptyWrapper()
        #print "out1"
        revN = model.maxVal-model.minVal
        return sb,eb*revN+model.minVal,self.model

      if(k % 50 ≡ 0):
         self.display2()
         self.model.evalBetter()
         count=0
    #print "out2"
    self.model.emptyWrapper()
    revN = model.maxVal-model.minVal
    return sb,eb*revN+model.minVal,self.model
```