```
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,score,printChar=’’):
    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
         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()
```

```
           break

         if random.random() > probLocalSearch:
             c = int(0 + (self.model.n−0)*random.random())
             solution[c]=model.neighbour(minR,maxR)
             self.display(score,"+"),
         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
             self.display(tempBestScore,"!"),
         self.display(score,"."),
         if(self.model.lives ≡ 0):
           self.display2()
           return bestSolution,bestScore,self.model
         if(j%50≡0):
            self.display2()
            self.model.evalBetter()

    return bestSolution,bestScore,self.model

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

class SA(SearchersBasic): #minimizing
  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)
    print "n: %d"%model.n
    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"
        return sb,eb,self.model
      if(k % 50 ≡ 0):
         self.display2()
         self.model.evalBetter()
       #  print "%f{%d}"%(sb,count),
         count=0
    #print "out2"
    self.model.emptyWrapper()
    return sb,eb,self.model

class GA(SearchersBasic):
  model = None
  minR=0
  maxR=0
  population={}
  random.seed(1)
  def __init__(self,modelName,displayS):
    self.model=modelName
    self.displayStyle=displayS
    self.crossoverRate = float(myoptions[’GA’][’crossOverRate’])
    self.mutationRate = 1/self.model.n
    self.elitismrank = int(myoptions[’GA’][’elitism’])
    self.generation = int(myoptions[’GA’][’generation’])

  def crossOver(self,listdaddy,listmommy):
    rate=self.crossoverRate
    #assert(len(listdaddy)==len(listmommy)),"Something’s messed up"
    if(random.random()<rate):
      minR,maxR=0,len(listdaddy)
      one = int(minR + random.random()*(maxR-minR))
      two = int(minR + random.random()*(maxR-minR))
      if(one≡two):two+=1
      newDaddy=listdaddy[:one]+listmommy[one:two]+listdaddy[two:]
      newMommy=listmommy[:one]+listdaddy[one:two]+listmommy[two:]
      return newDaddy,newMommy
    return listdaddy,listmommy

  def mutation(self,listdaddy,listmommy):
     rate=1#self.mutationRate
     #assert(len(listdaddy)==len(listmommy)),"Something’s messed up"
     if(random.random() < rate):
       #print "MUTATION"
       mutant = listdaddy[:]
       minR,maxR=0,min(len(listdaddy),len(listmommy))
       mutationE = int(minR + (random.random()*(maxR-minR)))
       mutationH = int(minR + (random.random()*(maxR-minR)))
       #print "++ %f %f"%(len(listdaddy),len(listmommy))
       #print ">> %f %f"%(mutationE,mutationH)
       mutant[mutationE]=listmommy[mutationH]
```

```
       return mutant
     return listdaddy

  #Changes a list of numbers to a stream of numbers
  #eg. [0.234,0.54,0.54325] -> [2345454325]
  def singleStream(self,listpoints):
     singlelist=[]
     for i in listpoints:
       tempstr = str(i)[2:]
       for x in tempstr:
         singlelist.append(x)
     #print singlelist
     return singlelist

  def generate(self):
    #http://stackoverflow.com/questions/4119070/
    #how-to-divide-a-list-into-n-equal-parts-python
    lol = lambda lst, sz: [lst[i:i+sz] for i in range(0, len(lst), sz)]
    model=self.model
    minR = model.minR
    maxR = model.maxR
    model.baseline(minR,maxR)
    temps1 = self.Roulette(self.population)
    temps2 = self.Roulette(self.population)
    #workaround: Bug: was getting e in temp2 so,
    #whenever I see anything other than 0-9
    #I replace it
    import re
    temps1 = re.sub(’[^0-9]’, ’’, temps1)
    temps2 = re.sub(’[^0-9]’, ’’, temps2)
    s1 = map(int, temps1)
    s2 = map(int, temps2)
    c1,c2=self.crossOver(s1,s2)
    m1 = self.mutation(c1,c2)
    m2 = self.mutation(c2,c1)
    #print len(m1),len(m2)
    #print model.n
    normalc1 = [int(’’.join(map(str,x)))/10**len(x) for x in lol(m1,int(len(m1)/
model.n))]
    normalc2 = [int(’’.join(map(str,x)))/10**len(x) for x in lol(m2,int(len(m1)/
model.n))]
    #normalc1 = map(lambda x:minR+x*(maxR-minR),normalc1)
    #normalc2 = map(lambda x:minR+x*(maxR-minR),normalc2)
    return normalc1,normalc2

  #http://stackoverflow.com/questions/10324015
  #/fitness-proportionate-selection-roulette-wheel-selection-in-python
  def Roulette(self,choices):
    max = sum(choices.values())
    pick = random.uniform(0, max)
    current = 0
    for key, value in choices.items():
        current += value
        if current > pick:
            return key

  def keyTransform(self,s):
    minR = self.model.minR
    maxR = self.model.maxR
    strs = self.singleStream(s)
    strs = (’’.join(map(str,strs)))
    fitness = self.model.evaluate(map(lambda x:minR+x*(maxR-minR),s))
    return strs,fitness

  def initialPopulation(self):
    model=self.model
    for i in xrange(50):
      s = [random.random() for z in range(0,model.n)]
      strs,fitness = self.keyTransform(s)
      self.population[strs]=fitness

  def elitism(self):
    rank = self.elitismrank
    #print len(self.population),
    #This controls whether this GA maximizes
    #or minimizes
    l = sorted(self.population.values())
    l = l[rank:]
    # TODO: not at all efficient
    for i in l:
      self.population = {key: value \
      for key, value in self.population.items() \
             if value is ¬ i}
    #print len(self.population)


  def evaluate(self):
    bestSolution=[]
    bestScore = 1e6
    done=False
    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)
    print "n: %d"%model.n
    self.initialPopulation()
    for x in xrange(self.generation):
      #print "Generation: %d"%x
      for i in xrange(20):
        s1,s2 = self.generate()
        #TODO: dirty
        strs,fitness = self.keyTransform(s1)
        self.population[strs]=fitness
        strs,fitness = self.keyTransform(s2)
        self.display(score=fitness)
        self.population[strs]=fitness
        if(fitness<bestScore):
          bestScore=fitness
          bestSolution=strs
      self.elitism()
      self.display2()

    print sorted(self.population.values())
    lol = lambda lst, sz: [lst[i:i+sz] \
    for i in range(0, len(lst), sz)]
    tempSolution = [int(’’.join(map(str,x)))/10**len(x)\
     for x in lol(bestSolution,int(len(bestSolution)/model.n))]
    print map(lambda x:minR+x*(maxR−minR),tempSolution)
```