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

sqrt=math.sqrt

class Log(): #only 1 attribute can be stored here
  def __init__(self):
    self.listing=[]
    self.history=[] #Would have the history
    self.historyhi,self.historylo,self.historyIndex=−1e10,1e10,0
    self.lo,self.hi,self.median,self.iqr=1e10,−1e10,0,0
    self.changed=True


  def add(self,num):
    if num≡None: return num
    self.listing.append(num)
    self.lo=min(self.lo,num)
    self.hi=max(self.hi,num)
    #print self.lo,self.hi
    self.changed=True

  def stats(self):
    temp=sorted(self.listing)
    n=len(temp)
    p=n//2
    if n%2 : return temp[p]
    q = max(0,(min(p+1,n)))
    self.iqr=temp[int(n*.75)] − temp[int(n*.25)]
    self.median=(temp[p]+temp[q])/2
    self.changed=False
    return self.median,self.iqr

  def empty(self):
    import copy
    self.history.append(self.listing)
    self.historyIndex+=1
    self.historylo=min(self.lo,self.historylo)
    self.historyhi=max(self.hi,self.historyhi)

    self.listing=[]
    self.lo,self.hi,self.median,self.iqr=1e6,−1e6,0,0
    self.changed=True

  def report(self):
    if self.changed ≡ False: return median,iqr
    return self.stats()



class ModelBasic(object):
  objf=None
  past =None #List of Logs
  present = None #List of Logs
  lives=None

  #From Dr. M's files: a12.py
  def a12slow(self,lst1,lst2):
    more = same = 0.0
    for x in sorted(lst1):
      for y in sorted(lst2):
        if   x≡y :
          same += 1
        elif x > y :
          more += 1
    return (more + 0.5*same) / (len(lst1)*len(lst2))
```

```


  """
  Given two logs, it would maintain states of lives etc
  """
  def better(self,past,present):
    betteriqr,bettermedian= False,False
    if(len(past.listing) ≡ 0 ): return(True,False)
    #if len(past.listing) == None: return (True,False)
    if(present.changed ≡ True): present.report()

    bettermedian = past.median > present.median
    if bettermedian ≡ True: return (True,self.a12slow(past.listing,present.listing)≤ myModeloptions['a12'])
    if past.median ≡ present.median:
       betteriqr = past.iqr > present.iqr
    return betteriqr,self.a12slow(past.listing,present.listing)≤ myModeloptions['a12']

  def evalBetter(self):
    better,same=[],[]
    for x in xrange(self.objf):
      tempbetter,tempsame=self.better(self.past[x],self.present[x])
      better.append(tempbetter)
      same.append(tempsame)

    import operator
    if(reduce(operator.and_,same)≡True):
      self.lives−=1
      #print "−−−−−−−−−−−−−DIE"
    elif(reduce(operator.or_,better)≡True): #need to check!
      pass
    else:
      self.lives−=1
      #print "−−−−−−−−−−−−−DIE"
    self.emptyWrapper()

  def emptyWrapper(self):
    #print "emptyWrapper"
    for x in xrange(self.objf):
      self.past[x].empty()
      import copy
      #http://stackoverflow.com/questions/184643/
      #what−is−the−best−way−to−copy−a−list
      self.past[x].listing = copy.copy(self.present[x].listing)
      self.past[x].listing = copy.copy(self.present[x].listing)
      self.past[x].lo = self.present[x].lo
      self.past[x].hi = self.present[x].hi
      self.present[x].empty()


  def returnMin(self,num):
    if(num<self.minVal):
      self.minVal=num
      return num
    else:
      return self.minVal

  def returnMax(self,num):
    if(num>self.maxVal):
      self.maxVal=num
      return num
    else:
      return self.maxVal

  def addWrapper(self,listpoint):#list of objective scores
    #len(listpoint) should be equal to objective function(self.objf)
    if(listpoint≡None): return None
    for x in xrange(len(listpoint)):
      self.present[x].add(listpoint[x])
      #print self.present[x].listing

  def evaluate(self,listpoint):
    #print "EVALUATE"
    temp=[]
    for x in xrange(0,self.objf):
       callName = "f"+str(x+1)
       #exec(getattr(self, callName)(listpoint))
       temp.append(getattr(self, callName)(listpoint))

    self.addWrapper(temp)
    #print temp
    energy= np.sum(temp)

    #f1 = self.f1(listpoint)
    #f2 = self.f2(listpoint)
    #self.presentLogf1.add(f1)
    #self.presentLogf2.add(f2)
    #energy = f1+f2
    return (energy-self.minVal)/(self.maxVal-self.minVal)



  def  neighbour(self,minN,maxN):
    return minN + (maxN-minN)*random.random()

class Fonseca(ModelBasic):
  def __init__(self,minR=-4,maxR=4,n=3,objf=2):
    self.minR=minR
    self.maxR=maxR
    self.n=n
    self.minVal=10000000
    self.maxVal=-1e6
    self.objf=objf
    self.past = [Log() for count in xrange(objf)]
    self.present = [Log() for count in xrange(objf)]
    self.lives=myModeloptions['Lives']

  def f1(self,listpoint):
    n=len(listpoint)
    rootn=(n**0.5)**-1
    sum=0
    for i in range(0,n):
        sum+=(listpoint[i]-rootn)**2
    return (1 - math.exp(-sum))

  def f2(self,listpoint):
    n=len(listpoint)
    rootn=(n**0.5)**-1
    sum=0
    for i in range(0,n):
        sum+=(listpoint[i]+rootn)**2
    return (1 - math.exp(-sum))

  def info(self):
    return "Fonseca~"

  def baseline(self,minR,maxR):
    for x in range(0,100000):
      solution = [(minR + random.random()*(maxR-minR)) for z in range(0,3)]
      self.returnMax(self.f1(solution)+ self.f2(solution))
      self.returnMin(self.f1(solution)+ self.f2(solution))


class Kursawe(ModelBasic):
  def __init__(self,minR=-5,maxR=5,n=3,objf=2):
    self.minR=minR
    self.maxR=maxR
    self.n=n
    self.minVal=10000000
    self.maxVal=-1e6
    self.objf=objf
    self.past = [Log() for count in xrange(objf)]
    self.present = [Log() for count in xrange(objf)]
    self.lives=myModeloptions['Lives']
```

```
  def f1(self,listpoint):
    n=len(listpoint)
    #inspired by 'theisencr'
    return np.sum([-10*math.exp(-0.2*(np.sqrt(listpoint[i]**2 + listpoint[i+1]**2))) for i in range (0, n-1)])
    return sum

  def f2(self,listpoint):
    a=0.8
    b=3
    n=len(listpoint)
    #inspired by 'theisencr'
    return np.sum([math.fabs(listpoint[i])**a + 5*np.sin(listpoint[i])**b for i in range (0, n)])

  def info(self):
    return "Kursawe~"

  def baseline(self,minR,maxR):
    for x in range(0,50000):
      solution = [(minR + random.random()*(maxR-minR)) for z in range(0,3)]
      self.returnMax(self.f1(solution)+ self.f2(solution))
      self.returnMin(self.f1(solution)+ self.f2(solution))

class ZDT1(ModelBasic):
  maxVal=-10000
  minVal=10000

  def __init__(self,minR=0,maxR=1,n=30,objf=2):
    self.minR=minR
    self.maxR=maxR
    self.n=n
    self.objf=objf
    self.past = [Log() for count in xrange(objf)]
    self.present = [Log() for count in xrange(objf)]
    self.lives=myModeloptions['Lives']

  def f1(self,lst):
    assert(len(lst)≡self.n),"Something's Messed up"
    return lst[0]

  def gx(self,lst):
    n=self.n
    assert(len(lst) ≡ n),"Something's Messed up"
    return (1+ 9*np.sum([lst[i] for i in range(1,n)])/(n-1))

  def f2(self,lst):
    n=self.n
    assert(len(lst)≡n),"Something's Messed up"
    gx=self.gx(lst)
    assert(gx≠0),"Ouch! it hurts"
    return gx * (1- sqrt(lst[0]/gx))


  def baseline(self,minR=0,maxR=1):
    for x in range(0,90000):
      solution = [(minR + random.random()*(maxR-minR)) for z in range(0,30)]
      self.returnMax(self.f1(solution)+ self.f2(solution))
      self.returnMin(self.f2(solution)+ self.f2(solution))

  def info(self):
    return "ZDT1~"


class Schaffer(ModelBasic):

  def __init__(self,minR=-1e4,maxR=1e4,n=1,objf=2):
    self.minR=minR
    self.maxR=maxR
    self.n=n
    self.minVal=10000000
    self.maxVal=-1e6
    self.objf=objf
```

```
        self.past = [Log() for count in xrange(objf)]
        self.present = [Log() for count in xrange(objf)]
        self.lives=myModeloptions['Lives']
     """
    def evaluate(self,listpoint):
      assert(len(listpoint) == 1),"Something's Messed up"
      var=listpoint[0]
      f1 = var**2
      f2 = (var−2)**2
      self.presentLogf1.add(f1)
      self.presentLogf2.add(f2)
      rawEnergy = f1+f2
      energy = (rawEnergy −self.minVal)/(self.maxVal−self.minVal)
      return energy
    """
    def f1(self,lst):
      return lst[0]**2

    def f2(self,lst):
      return (lst[0]−2)**2

    def info(self):
      return "Schaffer~"

    def baseline(self,minR,maxR):
      low = self.minR
      high = self.maxR
      for index in range(0,1000000):
        inputRand =(low + (high−low)*random.random())
        #print "inputRand: %s"%inputRand
        temp = (inputRand**2 +(inputRand−2)**2)
        self.minVal=self.returnMin(temp)
        self.maxVal=self.returnMax(temp)
      print("Max: %d Min: %d"%(self.maxVal,self.minVal))

  class ZDT3(ModelBasic):

    def __init__(self,minR=0,maxR=1,n=30,objf=2):
      self.minR=minR
      self.maxR=maxR
      self.n=n
      self.minVal=1e6
      self.maxVal=−1e6
      self.objf=objf
      self.past = [Log() for count in xrange(objf)]
      self.present = [Log() for count in xrange(objf)]
      self.lives=myModeloptions['Lives']

    def f1(self,listpoint):
      return listpoint[0];

    def gx(self,listpoint):
      return 1+((9/29)*sum([listpoint[i] for i in range(1,len(listpoint))]))

    def hx(self,listpoint):
      temp2 = (self.f1(listpoint)/self.gx(listpoint))**0.5
      temp32 = math.sin(10*math.pi*self.f1(listpoint))
      temp3 = (self.f1(listpoint)/self.gx(listpoint))* temp32
      return 1−temp2−temp3

    def f2(self,listpoint):
      return self.gx(listpoint)*self.hx(listpoint)

    def baseline(self,minR,maxR):
      for x in range(0,180000):
        solution = [(self.minR + random.random()*(self.maxR−self.minR)) for z in range(0,30)]
        self.returnMax(self.f1(solution)+ self.f2(solution))
        self.returnMin(self.f1(solution)+ self.f2(solution))

    def info(self):
      return "ZDT3~"

  class Viennet(ModelBasic):
```

```
    def __init__(self,minR=−3,maxR=3,n=2,objf=3):
      self.minR=minR
      self.maxR=maxR
      self.n=n
      self.minVal=1e6
      self.maxVal=−1e6
      self.objf=objf
      self.past = [Log() for count in xrange(objf)]
      self.present = [Log() for count in xrange(objf)]
      self.lives=myModeloptions['Lives']
      """
      self.pastLogf1 = Log()
      self.pastLogf2 = Log()
      self.pastLogf3 = Log()     #I am sorry this is crude
      self.presentLogf1 = Log()
      self.presentLogf2 = Log()
      self.presentLogf3 = Log()     #I am sorry this is crude
      """


    def f1(self,listpoint):
      x=listpoint[0]
      y=listpoint[1]
      return 0.5*(x**2+y**2)+math.sin(x**2+y**2)

    def f2(self,listpoint):
      x=listpoint[0]
      y=listpoint[1]
      temp1=(3*x−2*y+4)**2/8
      temp2=(x−y+1)**2/27
      return temp1+temp2+15

    def f3(self,listpoint):
      x=listpoint[0]
      y=listpoint[1]
      temp1=(x**2+y**2+1)**−1
      temp2=1.1*math.exp(−(x**2+y**2))
      return temp1+temp2
    """
    #@override
    def evalBetter(self):
      better1,same1=self.better(self.pastLogf1,self.presentLogf1)
      better2,same2=self.better(self.pastLogf2,self.presentLogf2)
      better3,same3=self.better(self.pastLogf3,self.presentLogf3)
      #print better1,same1,better2,same2

      if(same1&same2&same3 == True):
        self.lives−=1
      elif((better1 or better2 or better3) == True):
        pass
      else:
        self.lives−=1

      self.pastLogf1.empty()
      self.pastLogf2.empty()
      self.pastLogf3.empty()
      import copy #http://stackoverflow.com/questions/184643/what−is−the−best−way−to−copy−a−list
      self.pastLogf1.listing = copy.copy(self.presentLogf1.listing)
      self.pastLogf2.listing = copy.copy(self.presentLogf2.listing)
      self.pastLogf3.listing = copy.copy(self.presentLogf3.listing)
      self.presentLogf1.empty()
      self.presentLogf2.empty()
      self.presentLogf3.empty()

    def evaluate(self,listpoint):
      f1 = self.f1(listpoint)
      f2 = self.f2(listpoint)
      f3 = self.f3(listpoint)
      self.presentLogf1.add(f1)
      self.presentLogf2.add(f2)
      self.presentLogf3.add(f3)
      energy = f1+f2+f3
      return (energy−self.minVal)/(self.maxVal−self.minVal)
```

```
"""
def baseline(self,minR,maxR):
  for x in range(0,90000):
    solution = [(self.minR + random.random()*(self.maxR-self.minR)) for z in range(0,self.n)]
    self.returnMax(self.f1(solution)+ self.f2(solution)+self.f3(solution))
    self.returnMin(self.f1(solution)+ self.f2(solution)+self.f3(solution))
```