```
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=-9e10,9e10,0
    self.lo,self.hi,self.median,self.iqr=1e10,-1e10,0,0
    self.changed=True
    self.bestIndex=-1


  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)
    #print "Length: %d"%n
    p=n//2
    if(n%2=0) : return temp[p]
    q = max(0,(min(p+1,n-1)))
    #print "P:%d Q:%d"%(p,q)
    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 historyCopy(self):
    #print "historyCopy"
    import copy
    self.history.append(self.listing)
    self.historylo=min(self.lo,self.historylo)
    if(self.lo = self.historylo):self.bestIndex=self.historyIndex
    self.historyhi=max(self.hi,self.historyhi)
    self.historyIndex+=1
    #print self.historylo,self.historyhi

  def empty(self):
    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 self.median,self.iqr
    #print "report_____________________________",
    #print self.listing
    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):
    #print 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,same,bettermedian= False,False,False
    if(len(past.listing) = 0 ):
      return(True,True)
    #if len(past.listing) == None: return (True,False)
    if(present.changed = True):
      past.report()
      present.report()
    #print " pastMedian: %f presentMedian: %f"%(past.median,present.median)
    bettermedian = past.median > present.median
    betteriqr = past.iqr > present.iqr
    #print bettermedian,betteriqr
    return bettermedian,betteriqr

  def same(self,past,present):
    if(len(past.listing) = 0 ):
      self.emptyWrapper()
      return(False)
    return self.a12slow(past.listing,present.listing)≤ myModeloptions['a12']


  def evalBetter(self):
    def worsed():
      return  ((same     ∧ ¬ betterIqr) ∨
               (¬ same ∧ ¬ betterMed))
    def bettered():
      return  ¬ same ∧ betterMed
    out=False
    for x in xrange(self.objf):
      if(len(self.past[x].listing) ≠ 0):
        betterMed,betterIqr=self.better(self.past[x],self.present[x])
        same = self.same(self.past[x],self.present[x])
        #print "##############Worse %d"%worsed()
        #print "##############Better %d"%bettered()
        #print "asddddddddddddDD"
        #print betterMed,betterIqr,same
        if worsed():
          #print "---%d %d---ads--%d-----DIE"%(betterMed,betterIqr,x)
          self.lives-=1
          self.emptyWrapper()
          return False
        if bettered(): out = out ∨ True
      else:
        out=True
        break

    if(out = False):
      self.emptyWrapper()
      self.lives-=1
      #print "-------adas------DIE"
      return False
    self.emptyWrapper()
    return False

  def emptyWrapper(self):
    #print "emptyWrapper"
    for x in xrange(self.objf):
      self.past[x].historyCopy()
      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 "&&&&&&&&&&&&&&&&&&&&&&&&&&"
      #print listpoint[x]

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

    self.addWrapper(temp)
    #print temp
    energy= np.sum(temp)
    #print (energy-self.minVal)/(self.maxVal-self.minVal)
    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']
      self.functionDict = {}
      self.functionDict["f1"]="f1"
      self.functionDict["f2"]="f2"


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

  def f2(self,listpoint,num=0):
    n=len(listpoint)
    rootn=(n**0.5)**-1
    sum=0
    for i in range(0,n):
        sum+=(listpoint[i]+1/rootn)**2
    return (1 - np.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']
    self.functionDict = {}
    self.functionDict["f1"]="f1"
    self.functionDict["f2"]="f2"



  def f1(self,listpoint,num=0):
    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,num=0):
    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']
    self.functionDict = {}
    self.functionDict["f1"]="f1"
    self.functionDict["f2"]="f2"


  def f1(self,lst,num=0):
    assert(len(lst)=self.n),"Something's Messed up %d"%len(lst)
    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,num=0):
    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']
    self.functionDict = {}
    self.functionDict["f1"]="f1"
    self.functionDict["f2"]="f2"

  """
  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,num=0):
    return lst[0]**2

  def f2(self,lst,num=0):
    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']
    self.functionDict = {}
    self.functionDict["f1"]="f1"
    self.functionDict["f2"]="f2"

  def f1(self,listpoint,num=0):
    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,num=0):
    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,num=0):
    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.functionDict = {}
        self.functionDict["f1"]="f1"
        self.functionDict["f2"]="f2"
        self.functionDict["f3"]="f3"
        """
        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,num=0):
        x=listpoint[0]
        y=listpoint[1]
        return 0.5*(x**2+y**2)+math.sin(x**2+y**2)

      def f2(self,listpoint,num=0):
        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,num=0):
        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

      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))

    class DTLZ7(ModelBasic):
      def __init__(self,minR=0,maxR=1,objf=20,n=39,k=20):
        self.minR=minR
        self.maxR=maxR
        self.n=n
        self.k=k
        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']
        assert(self.k == self.n-self.objf+1),"Something's Messed up"
        self.functionDict = {}
        for i in xrange(objf-1):
          temp = "f"+str(i+1)
          self.functionDict[temp]="fi"
        temp="f"+str(objf)
        self.functionDict[temp]="fcrazy"


      def fi(self,listpoints,num):
        return listpoints[num-1]

      def fcrazy(self,listpoints,num):
        return (1+self.g(listpoints)*self.h(listpoints))

      def g(self,listpoints):
        summ=0
        try:
          #print "len of listpoints %d"%len(listpoints)
          for i in range(self.objf,self.n):
            #print i
            summ+=listpoints[i]
          return(1+9*summ/self.k)
        except:
          print i,len(listpoints)
          raise Exception("ERROR")

      def h(self,listpoints):
        g=self.g(listpoints)
        summ=0
        for i in range(0,self.objf):
          summ+=listpoints[i]/(1+g) * (1+math.sin(3*math.pi*listpoints[i]))
        return (self.objf-summ)

      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)]
          result=0
          for i in xrange(self.objf):
            temp="f"+str(i+1)
            callName = self.functionDict[temp]
            result+=int(getattr(self, callName)(solution,i+1))
          self.returnMax(result)
          self.returnMin(result)

    class Schwefel(ModelBasic):
      def __init__(self,minR=-math.pi,maxR=math.pi,objf=1,n=10):
```



```
        self.minR=minR
        self.maxR=maxR
        self.n=n
        self.f_bias=-460
        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.functionDict = {}
        self.functionDict["f1"]="f1"
        randInt = lambda x: random.randint(-x,x)
        randFloat = lambda x: random.uniform(-x,x)
        self.A = [[randInt(100) for _ in xrange(self.n)] for _ in xrange(self.n)]
        self.B = [[randInt(100) for _ in xrange(self.n)] for _ in xrange(self.n)]
        self.alpha = [randFloat(math.pi) for _ in xrange(self.n)]

      def f1(self,listpoints,num=0):
        return np.sum([(self.MA(n) - self.MB(listpoints,n))**2 for n in xrange(self.n)]) + self.f_bias

      def MA(self,n):
        return np.sum(self.A[n][j]*math.sin(self.alpha[j])+self.B[n][j]*math.cos(self.alpha[j]) for j in xrange(self.n))

      def MB(self,x,n):
        return np.sum([self.A[n][j]*math.sin(s) + self.B[n][j]*math.cos(s) for j,s in enumerate(x)])

      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)]
          result=0
          for i in xrange(self.objf):
            temp="f"+str(i+1)
            callName = self.functionDict[temp]
            result+=int(getattr(self, callName)(solution,i+1))
          self.returnMax(result)
          self.returnMin(result)
```