```
"""

## Hyptotheis Testing Stuff


### Standard Stuff

#### Standard Headers

"""
from __future__ import division
import sys,random,math
sys.dont_write_bytecode = True
"""

#### Standard Utils

"""
class o():
  "Anonymous container"
  def __init__(i,**fields) :
    i.override(fields)
  def override(i,d): i.__dict__.update(d); return i
  def __repr__(i):
    d = i.__dict__
    name = i.__class__.__name__
    return name+’{’+’ ’.join([’:%s %s’ % (k,pretty(d[k]))
                     for k in i.show()])+ ’}’
  def show(i):
    return [k for k in sorted(i.__dict__.keys())
            if not "_" in k]
"""

Misc functions:

"""
rand = random.random
any  = random.choice
seed = random.seed
exp  = lambda n: math.e**n
ln   = lambda n: math.log(n,math.e)
g    = lambda n: round(n,2)

def median(lst,ordered=False):
  if not ordered: lst= sorted(lst)
  n = len(lst)
  p = n//2
  if n % 2: return lst[p]
  q = p − 1
  q = max(0,min(q,n))
  return (lst[p] + lst[q])/2

def msecs(f):
  import time
  t1 = time.time()
  f()
  return (time.time() − t1) * 1000

def pairs(lst):
  "Return all pairs of items i,i+1 from a list."
  last=lst[0]
  for i in lst[1:]:
    yield last,i
    last = i

def xtile(lst,lo=0,hi=100,width=50,
             chops=[0.1 ,0.3,0.5,0.7,0.9],
             marks=["−" ," "," ","−"," "],
             bar="|",star="*",show=" %3.0f"):
  """The function _xtile_ takes a list of (possibly)
  unsorted numbers and presents them as a horizontal
  xtile chart (in ascii format). The default is a
  contracted _quintile_ that shows the
```

```
  10,30,50,70,90 breaks in the data (but this can be
  changed− see the optional flags of the function).
  """
  def pos(p)   : return ordered[int(len(lst)*p)]
  def place(x) :
    return int(width*float((x − lo))/(hi − lo+0.00001))
  def pretty(lst) :
    return ’, ’.join([show % x for x in lst])
  ordered = sorted(lst)
  lo      = min(lo,ordered[0])
  hi      = max(hi,ordered[−1])
  what    = [pos(p)   for p in chops]
  where   = [place(n) for n in  what]
  out     = [" "] * width
  for one,two in pairs(where):
    for i in range(one,two):
      out[i] = marks[0]
    marks = marks[1:]
  out[int(width/2)]    = bar
  out[place(pos(0.5))] = star
  return ’(’+’’.join(out) +  ")," +  pretty(what)

def _tileX() :
  import random
  random.seed(1)
  nums = [random.random()**2 for _ in range(100)]
  print xtile(nums,lo=0,hi=1.0,width=25,show=" %5.2f")
"""

### Standard Accumulator for Numbers

Note the _lt_ method: this accumulator can be sorted by median values.

Warning: this accumulator keeps _all_ numbers. Might be better to use
a bounded cache.

"""
class Num:
  "An Accumulator for numbers"
  def __init__(i,name,inits=[]):
    i.n = i.m2 = i.mu = 0.0
    i.all=[]
    i._median=None
    i.name = name
    i.rank = 0
    for x in inits: i.add(x)
  def s(i)       : return (i.m2/(i.n − 1))**0.5
  def add(i,x):
    i._median=None
    i.n   += 1
    i.all += [x]
    delta  = x − i.mu
    i.mu  += delta*1.0/i.n
    i.m2  += delta*(x − i.mu)
  def __add__(i,j):
    return Num(i.name + j.name,i.all + j.all)
  def quartiles(i):
    def p(x) : return int(g(xs[x]))
    i.median()
    xs = i.all
    n  = int(len(xs)*0.25)
    return p(n) , p(2*n) , p(3*n)
  def median(i):
    if not i._median:
      i.all = sorted(i.all)
      i._median=median(i.all)
    return i._median
  def __lt__(i,j):
    return i.median() < j.median()
  def spread(i):
    i.all=sorted(i.all)
    n1=i.n*0.25
    n2=i.n*0.75
    if len(i.all) <= 1:
      return 0
    if len(i.all) == 2:
      return i.all[1] − i.all[0]
    else:
      return i.all[int(n2)] − i.all[int(n1)]


"""

### The A12 Effect Size Test

"""
def a12slow(lst1,lst2):
  "how often is x in lst1 more than y in lst2?"
  more = same = 0.0
  for x in lst1:
    for y in lst2:
      if    x == y : same += 1
      elif  x >  y : more += 1
  x= (more + 0.5*same) / (len(lst1)*len(lst2))
  return x

def a12(lst1,lst2):
  "how often is x in lst1 more than y in lst2?"
  def loop(t,t1,t2):
    while t1.j < t1.n and t2.j < t2.n:
      h1 = t1.l[t1.j]
      h2 = t2.l[t2.j]
      h3 = t2.l[t2.j+1] if t2.j+1 < t2.n else None
      if h1>  h2:
        t1.j  += 1; t1.gt += t2.n − t2.j
      elif h1 == h2:
        if h3 and h1 > h3 :
            t1.gt += t2.n − t2.j  − 1
        t1.j  += 1; t1.eq += 1; t2.eq += 1
      else:
        t2,t1  = t1,t2
    return t.gt*1.0, t.eq*1.0
  #−−−−−−−−−−−−−−−−−−−−−−−−−−
  lst1 = sorted(lst1, reverse=True)
  lst2 = sorted(lst2, reverse=True)
  n1   = len(lst1)
  n2   = len(lst2)
  t1   = o(l=lst1,j=0,eq=0,gt=0,n=n1)
  t2   = o(l=lst2,j=0,eq=0,gt=0,n=n2)
  gt,eq= loop(t1, t1, t2)
  return gt/(n1*n2) + eq/2/(n1*n2)

def _a12():
  def f1(): return a12slow(l1,l2)
  def f2(): return a12(l1,l2)
  for n in [100,200,400,800,1600,3200,6400]:
    l1 = [rand() for _ in xrange(n)]
    l2 = [rand() for _ in xrange(n)]
    t1 = msecs(f1)
    t2 = msecs(f2)
    print n, g(f1()),g(f2()),int((t1/t2))


"""Output:

‘‘‘‘
n   a12(fast)       a12(slow)       tfast / tslow
--- --------------- -------------- --------------
100  0.53           0.53               4
200  0.48           0.48               6
400  0.49           0.49              28
800  0.5            0.5               26
1600 0.51           0.51              72
3200 0.49           0.49             109
6400 0.5            0.5              244
‘‘‘‘
```

```


## Non−Parametric Hypothesis Testing

The following _bootstrap_ method was introduced in
1979 by Bradley Efron at Stanford University. It
was inspired by earlier work on the
jackknife.
Improved estimates of the variance were [developed later][efron01].

[efron01]: http://goo.gl/14n8Wf "Bradley Efron and R.J. Tibshirani. An Introduction to the Bootstrap (Chapman & Hall/CRC Monographs on Statistics & Applied Probability), 1993"


To check if two populations _(y0,z0)_
are different, many times sample with replacement
from both to generate _(y1,z1), (y2,z2), (y3,z3)_.. etc.

"""
def sampleWithReplacement(lst):
  "returns a list same size as list"
  def any(n)  : return random.uniform(0,n)
  def one(lst): return lst[ int(any(len(lst))) ]
  return [one(lst) for _ in lst]
"""


Then, for all those samples,
 check if some *testStatistic* in the original pair
hold for all the other pairs. If it does more than (say) 99%
of the time, then we are 99% confident in that the
populations are the same.

In such a _bootstrap_ hypothesis test, the *some property*
is the difference between the two populations, muted by the
joint standard deviation of the populations.

"""
def testStatistic(y,z):
    """Checks if two means are different, tempered
     by the sample size of ’y’ and ’z’"""
    tmp1 = tmp2 = 0
    for y1 in y.all: tmp1 += (y1 − y.mu)**2
    for z1 in z.all: tmp2 += (z1 − z.mu)**2
    s1    = (float(tmp1)/(y.n − 1))**0.5
    s2    = (float(tmp2)/(z.n − 1))**0.5
    delta = z.mu − y.mu
    if s1+s2:
      delta =  delta/((s1/y.n + s2/z.n)**0.5)
    return delta
"""

The rest is just details:

+ Efron advises
  to make the mean of the populations the same (see
  the _yhat,zhat_ stuff shown below).
+ The class _total_ is a just a quick and dirty accumulation class.
+ For more details see [the Efron text][efron01].

"""
def bootstrap(y0,z0,conf=0.01,b=1000):
  """The bootstrap hypothesis test from
     p220 to 223 of Efron’s book ’An
    introduction to the boostrap."""
  class total():
    "quick and dirty data collector"
    def __init__(i,some=[]):
      i.sum = i.n = i.mu = 0 ; i.all=[]
      for one in some: i.put(one)
    def put(i,x):
      i.all.append(x);
      i.sum +=x; i.n += 1; i.mu = float(i.sum)/i.n
    def __add__(i1,i2): return total(i1.all + i2.all)
  y, z   = total(y0), total(z0)
  x      = y + z
  tobs   = testStatistic(y,z)
  yhat   = [y1 - y.mu + x.mu for y1 in y.all]
  zhat   = [z1 - z.mu + x.mu for z1 in z.all]
  bigger = 0.0
  for i in range(b):
    if testStatistic(total(sampleWithReplacement(yhat)),
                     total(sampleWithReplacement(zhat))) > tobs:
      bigger += 1
  return bigger / b < conf
"""

#### Examples

"""
def _bootstraped():
  def worker(n=1000,
             mu1=10,  sigma1=1,
             mu2=10.2, sigma2=1):
    def g(mu,sigma) : return random.gauss(mu,sigma)
    x = [g(mu1,sigma1) for i in range(n)]
    y = [g(mu2,sigma2) for i in range(n)]
    return n,mu1,sigma1,mu2,sigma2,\
        'different' if bootstrap(x,y) else 'same'
  # very different means, same std
  print worker(mu1=10, sigma1=10,
               mu2=100, sigma2=10)
  # similar means and std
  print worker(mu1= 10.1, sigma1=1,
               mu2= 10.2, sigma2=1)
  # slightly different means, same std
  print worker(mu1= 10.1, sigma1= 1,
               mu2= 10.8, sigma2= 1)
  # different in mu eater by large std
  print worker(mu1= 10.1, sigma1= 10,
               mu2= 10.8, sigma2= 1)
"""

Output:

''''
_bootstraped()

(1000, 10, 10, 100, 10, 'different')
(1000, 10.1, 1, 10.2, 1, 'same')
(1000, 10.1, 1, 10.8, 1, 'different')
(1000, 10.1, 10, 10.8, 1, 'same')
''''

Warning- the above took 8 seconds to generate since we used 1000 bootstraps.
As to how many bootstraps are enough, that depends on the data. There are
results saying 200 to 400 are enough but, since I am  suspicious man, I run it f
or 1000.

Which means the runtimes associated with bootstrapping is a significant issue.
To reduce that runtime, I avoid things like an all-pairs comparison of all treat
ments
(see below: Scott-knott).  Also, BEFORE I do the boostrap, I first run
the effect size test (and only go to bootstrapping in effect size passes:

"""
def different(l1,l2):
  #return bootstrap(l1,l2) and a12(l2,l1)
  return a12(l2,l1) and bootstrap(l1,l2)

"""

## Saner Hypothesis Testing

The following code, which you should use verbatim does the following:
```

```

+ All treatments are clustered into _ranks_. In practice, dozens
  of treatments end up generating just a handful of ranks.
+ The numbers of calls to the hypothesis tests are minimized:
    + Treatments are sorted by their median value.
    + Treatments are divided into two groups such that the
      expected value of the mean values _after_ the split is minimized;
    + Hypothesis tests are called to test if the two groups are truly difference
.
          + All hypothesis tests are non-parametric and include (1) effect size
tests
            and (2) tests for statistically significant numbers;
          + Slow bootstraps are executed  if the faster _A12_ tests are passed;

In practice, this means that the hypothesis tests (with confidence of say, 95%)
are called on only a logarithmic number of times. So...

+ With this method, 16 treatments can be studied using less than _&sum;<sub>1,2,
4,8,16</sub>log<sub>2</sub>i =15_ hypothesis tests  and confidence _0.99<sup>15<
/sup>=0.86_.
+ But if did this with the 120 all-pairs comparisons of the 16 treatments, we wo
uld have total confidence _0.99<sup>120</sup>=0.30.

For examples on using this code, see _rdivDemo_ (below).

"""
def scottknott(data,cohen=0.3,small=3, useA12=False,epsilon=0.01):
  """Recursively split data, maximizing delta of
  the expected value of the mean before and
  after the splits.
  Reject splits with under 3 items"""
  all  = reduce(lambda x,y:x+y,data)
  same = lambda l,r: abs(l.median() - r.median()) <= all.s()*cohen
  if useA12:
    same = lambda l, r:   not different(l.all,r.all)
  big  = lambda    n: n > small
  return rdiv(data,all,minMu,big,same,epsilon)

def rdiv(data,  # a list of class Nums
         all,   # all the data combined into one num
         div,   # function: find the best split
         big,   # function: rejects small splits
         same, # function: rejects similar splits
         epsilon): # small enough to split two parts
  """Looks for ways to split sorted data,
  Recurses into each split. Assigns a 'rank' number
  to all the leaf splits found in this way.
  """
  def recurse(parts,all,rank=0):
    "Split, then recurse on each part."
    cut,left,right = maybeIgnore(div(parts,all,big,epsilon),
                                 same,parts)
    if cut:
      # if cut, rank "right" higher than "left"
      rank = recurse(parts[:cut],left,rank) + 1
      rank = recurse(parts[cut:],right,rank)
    else:
      # if no cut, then all get same rank
      for part in parts:
        part.rank = rank
    return rank
  recurse(sorted(data),all)
  return data

def maybeIgnore((cut,left,right), same,parts):
  if cut:
    if same(sum(parts[:cut],Num('upto')),
            sum(parts[cut:],Num('above'))):
      cut = left = right = None
  return cut,left,right

def minMu(parts,all,big,epsilon):
  """Find a cut in the parts that maximizes
  the expected value of the difference in
  the mean before and after the cut.
  Reject splits that are insignificantly
  different or that generate very small subsets.
  """
  cut,left,right = None,None,None
  before, mu     =  0, all.mu
  for i,l,r in leftRight(parts,epsilon):
    if big(l.n) and big(r.n):
      n   = all.n * 1.0
      now = l.n/n*(mu- l.mu)**2 + r.n/n*(mu- r.mu)**2
      if now > before:
        before,cut,left,right = now,i,l,r
  return cut,left,right

def leftRight(parts,epsilon=0.01):
  """Iterator. For all items in ’parts’,
  return everything to the left and everything
  from here to the end. For reasons of
  efficiency, take a first pass over the data
  to pre-compute and cache right-hand-sides
  """
  rights = {}
  n = j = len(parts) - 1
  while j > 0:
    rights[j] = parts[j]
    if j < n: rights[j] += rights[j+1]
    j -=1
  left = parts[0]
  for i,one in enumerate(parts):
    if i> 0:
      if parts[i]._median - parts[i-1]._median > epsilon:
        yield i,left,rights[i]
      left += one
"""

## Putting it All Together

Driver for the demos:

"""
def rdivDemo(data):
  def z(x):
    return int(100 * (x - lo) / (hi - lo + 0.00001))
  data = map(lambda lst:Num(lst[0],lst[1:]),
             data)
  print ""
  ranks=[]
  for x in scottknott(data,useA12=True):
    ranks += [(x.rank,x.median(),x)]
  all=[]
  for _,__,x in sorted(ranks): all += x.all
  all = sorted(all)
  lo, hi = all[0], all[-1]
  line = "----------------------------------------------------"
  last = None
  print  (’%4s , %12s ,    %s   , %4s ’ % \
               (’rank’, ’name’, ’med’, ’iqr’))+ "\n"+ line
  for _,__,x in sorted(ranks):
    q1,q2,q3 = x.quartiles()
    print  (’%4s , %12s ,    %4s  ,  %4s ’ % \
                 (x.rank+1, x.name, q2, q3 - q1))  + \
              xtile(x.all,lo=lo,hi=hi,width=30,show="%5.2f")
    last = x.rank
"""

The demos:

"""
def rdiv0():
  rdivDemo([
        ["x1",0.34, 0.49, 0.51, 0.6],
```

```
        ["x2",6,  7,  8,  9] ])
"""

’’’’
rank ,         name ,    med   ,  iqr
----------------------------------------------------
   1 ,           x1 ,      51  ,    11 (*              |              ), 0.34,
0.49,  0.51,  0.51,  0.60
   2 ,           x2 ,     800  ,   200 (               |   ----   *-- ), 6.00,
7.00,  8.00,  8.00,  9.00
’’’’

"""
def rdiv1():
  rdivDemo([
        ["x1",0.1,  0.2,  0.3,  0.4],
        ["x2",0.1,  0.2,  0.3,  0.4],
        ["x3",6,  7,  8,  9] ])
"""

’’’’
rank ,         name ,    med   ,  iqr
----------------------------------------------------
   1 ,           x1 ,      30  ,    20 (*              |              ), 0.10,
0.20,  0.30,  0.30,  0.40
   1 ,           x2 ,      30  ,    20 (*              |              ), 0.10,
0.20,  0.30,  0.30,  0.40
   2 ,           x3 ,     800  ,   200 (               |   ----   *-- ), 6.00,
7.00,  8.00,  8.00,  9.00
’’’’

"""
def rdiv2():
  rdivDemo([
        ["x1",0.34, 0.49, 0.51, 0.6],
        ["x2",0.6,  0.7,  0.8,  0.9],
        ["x3",0.15, 0.25, 0.4,  0.35],
        ["x4",0.6,  0.7,  0.8,  0.9],
        ["x5",0.1,  0.2,  0.3,  0.4] ])
"""

’’’’
rank ,         name ,    med   ,  iqr
----------------------------------------------------
   1 ,           x5 ,      30  ,    20 (---    *---    |              ), 0.10,
0.20,  0.30,  0.30,  0.40
   1 ,           x3 ,      35  ,    15 ( ----    *-    |              ), 0.15,
0.25,  0.35,  0.35,  0.40
   2 ,           x1 ,      51  ,    11 (        ------ *--            ), 0.34,
0.49,  0.51,  0.51,  0.60
   3 ,           x2 ,      80  ,    20 (               |  ----    *-- ), 0.60,
0.70,  0.80,  0.80,  0.90
   3 ,           x4 ,      80  ,    20 (               |  ----    *-- ), 0.60,
0.70,  0.80,  0.80,  0.90
’’’’

"""
def rdiv3():
  rdivDemo([
      ["x1",101, 100, 99,   101,  99.5],
      ["x2",101, 100, 99,   101, 100],
      ["x3",101, 100, 99.5, 101,  99],
      ["x4",101, 100, 99,   101, 100] ])
"""

’’’’
rank ,         name ,    med   ,  iqr
----------------------------------------------------
   1 ,           x1 ,    10000  ,   150 (-------       *|              ),99.00,
99.50, 100.00, 101.00, 101.00
   1 ,           x2 ,    10000  ,   100 (--------------*|              ),99.00,
100.00, 100.00, 101.00, 101.00
   1 ,           x3 ,    10000  ,   150 (-------       *|              ),99.00,
99.50, 100.00, 101.00, 101.00
   1 ,           x4 ,    10000  ,   100 (--------------*|              ),99.00,
100.00, 100.00, 101.00, 101.00
’’’’

"""
def rdiv4():
  rdivDemo([
      ["x1",11,12,13],
      ["x2",14,31,22],
      ["x3",23,24,31],
      ["x5",32,33,34]])
"""

’’’’
rank ,         name ,    med   ,  iqr
----------------------------------------------------
   1 ,           x1 ,    1100  ,     0 ( *             |              ),11.00, 1
1.00, 12.00, 13.00, 13.00
   1 ,           x2 ,    1400  ,     0 (              *|              ),14.00, 1
4.00, 22.00, 31.00, 31.00
   2 ,           x3 ,    2300  ,     0 (               |*             ),23.00, 2
3.00, 24.00, 31.00, 31.00
   2 ,           x5 ,    3200  ,     0 (               |            * ),32.00, 3
2.00, 33.00, 34.00, 34.00
’’’’

"""
def rdiv5():
  rdivDemo([
      ["x1",11,11,11],
      ["x2",11,11,11],
      ["x3",11,11,11]])
"""

’’’’
rank ,         name ,    med   ,  iqr
----------------------------------------------------
   1 ,           x1 ,    1100  ,     0 (*              |              ),11.00, 1
1.00, 11.00, 11.00, 11.00
   1 ,           x2 ,    1100  ,     0 (*              |              ),11.00, 1
1.00, 11.00, 11.00, 11.00
   1 ,           x3 ,    1100  ,     0 (*              |              ),11.00, 1
1.00, 11.00, 11.00, 11.00
’’’’

"""
def rdiv6():
  rdivDemo([
      ["x1",11,11,11],
      ["x2",11,11,11],
      ["x4",32,33,34,35]])
"""

’’’’
rank ,         name ,    med   ,  iqr
----------------------------------------------------
   1 ,           x1 ,    1100  ,     0 (*              |              ),11.00, 1
1.00, 11.00, 11.00, 11.00
   1 ,           x2 ,    1100  ,     0 (*              |              ),11.00, 1
1.00, 11.00, 11.00, 11.00
   2 ,           x4 ,    3400  ,   200 (               |          - * ),32.00, 3
3.00, 34.00, 34.00, 35.00
’’’’

"""
def rdiv7():
  rdivDemo([
    ["x1"] +  [rand()**0.5 for _ in range(256)],
    ["x2"] +  [rand()**2   for _ in range(256)],
    ["x3"] +  [rand()      for _ in range(256)]
  ])
"""
```

```
’’’’
rank ,         name ,    med   ,  iqr
----------------------------------------------------
   1 ,           x2 ,      25  ,    50 (--     *      -|---------     ), 0.01,
0.09,  0.25,  0.47,  0.86
   2 ,           x3 ,      49  ,    47 (  ------      *|   -------    ), 0.08,
0.29,  0.49,  0.66,  0.89
   3 ,           x1 ,      73  ,    37 (         ------|-    *   ---  ), 0.32,
0.57,  0.73,  0.86,  0.95
’’’’

"""

def _rdivs():
  seed(1)
  print([k for k in globals().keys() if k.startswith(’rdiv’)])
  for fname in [’rdiv’ + str(n) for n in range(9)]:
    if fname in globals().keys():
      globals()[fname]()

def rdiv8():
    rdivDemo([
        [’TPBs’, 208, 176, 321, 128, 128],
        [’phil’, 688, 346, 290, 524],
        ["’zines", 28, 76, 32, 64],
        [’comp’, 398, 312, 361, 436, 316]
    ])

rdiv8()
```

```
from __future__ import division, print_function

from datetime import datetime
import random, time

from witschey.models import Schaffer, Fonseca, Kursawe
from witschey.models import ZDT1, ZDT3, Viennet3
from witschey.searchers import SimulatedAnnealer, MaxWalkSat
from witschey.log import NumberLog

def run(r=20, seed=10, text_report=False):
    print(datetime.now())
    for klass in (Schaffer,):
    # for klass in (Schaffer, Fonseca, Kursawe, ZDT1, ZDT3, Viennet3):
        xtiles = []
        print("\n", klass.__name__, sep=’’)
        print(’−’ * 50)
        # for searcher in (SimulatedAnnealer,):
        for searcher in (SimulatedAnnealer, MaxWalkSat):
            random.seed(seed)
            n = NumberLog(max_size=None)
            times = NumberLog(max_size=None)
            print(searcher.__name__)
            for _ in range(r):
                start_time = time.clock()
                s = searcher(klass())
                out = s.run(text_report=text_report)
                times += time.clock() − start_time
                n += out.best
            print(s.spec.to_str(sep=’: ’))
            if text_report:
                print(out.report)

            if hasattr(out, ’era_logs’):
                for fname, logs in sorted(out.era_logs.iteritems()):
                    print(’<’, fname)
                    for era, log in logs.iteritems():
                        print(era, log.xtile(width=20), sep=’\t’)

            print(’Best: {: .4f}’.format(n.mean()))
            print(’total time: {:.3f}s’.format(times.total()),
                ’mean time: {:.3f}s’.format(times.mean()), sep=’\t’)

            print(n.xtile(width=30), sep=’\n’)
            print(’\n’)
        print(’=’ * 50 + ’\n’, ’=’ * 50, sep=’’)

if __name__ == ’__main__’:
    run(r=1, seed=1, text_report=True)
```

```
from __future__ import division, print_function, unicode_literals

import json, random, functools, sys, math

def pretty_input(t):
    float_format = lambda x: ’{: .2f}’.format(x)
    str_tuple = tuple(float_format(x).encode(sys.stdout.encoding) for x in t)
    return ’, ’.join(s for s in str_tuple)

def pairs(xs):
    for p in zip(xs[:−1], xs[1:]):
        yield p

class memo():
    ’’’adapted from https://github.com/timm/sbse14/wiki/basepy’’’
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    # from http://stackoverflow.com/a/15538391/3408454
    def to_JSON(self, indent=None):
        ’adapted from from http://stackoverflow.com/a/15538391/3408454’

        d = lambda o: o.__dict__
        return json.dumps(self, default=d, sort_keys=True, indent=indent)

    def to_str(self, depth=0, indent=4, sep=’\u2192’, d=None):
        return self._to_str(
            depth=depth,
            indent=indent,
            sep=sep,
            d = self.__dict__ if d is None else d)

    def _to_str(self, depth, indent, sep, d):
        after = []
        reps = []
        rv = ’’
        for k in sorted([s for s in d.keys() if s[0] != ’_’]):
            val = d[k]
            if isinstance(val, (memo, dict)):
                after.append(k)
            else:
                if callable(val):
                    val = val.__name__ + ’()’
                reps.append(’{}{}{}’.format(k, sep, val))
        rv += ’ ’ * depth * indent
        rv += ’, ’.join(reps)
        rv += ’\n’

        for k in after:
            rv += ’ ’ * depth * indent
            rv += ’{ ’
            rv += ’{}:\n’.format(k)
            k = d[k]
            k = k if isinstance(k, dict) else k.__dict__
            rv += self._to_str(depth=depth+1, indent=indent, sep=sep, d=k)
            rv += ’ ’ * depth * indent
            rv += ’}’
            rv += ’\n’

        return rv

def memoize(f):
    ’memoizer for single−arg functions’
    d = {}
    @functools.wraps(f)
    def wrapper(x):
        try:
            return d[x]
        except KeyError:
            d[x] = f(x)
            return d[x]

    return wrapper
```

```
@memoize
def memo_sqrt(x):
    return math.sqrt(x)

def tuple_replace(t, replace_at, value):
    return tuple(value if i == replace_at else v for i, v in enumerate(t))

def random_index(x):
    if isinstance(x, list):
        return random.randint(0, len(x) − 1)
    if isinstance(x, dict):
        return random.choice(x.keys)
    raise ValueError(’{} is not a list or dict’.format(x))

The = memo(
    Searcher=memo(era_length=50, log_eras=True),
    SimulatedAnnealer=memo(iterations=1000, p_mutation=1/3),
    MaxWalkSat=memo(iterations=1000, p_mutation=1/3))
```

```
"""## Log Stuff

Adapted from [Dr. Tim Menzies’ logging code](https://github.com/timm/sbse14/blob
/master/log.py).

Logs are places to store records of past events. There are two types of logs:

+ _Num_ : for numbers
+ _Sym_ : for everything else.

Those logs can be queried to find e.g. the highest
and lowest value of the number seen so far. Alternatively,
they can be queried to return values at the same probability
as the current log contents.

### Max Log Size

To avoid logs consuming all memory, logs store at
most _The.cache.keep_ entries (e.g. 128):

+ If more
than that number of entries arrive, then some old
entry (selected at random) will be deleted.
+ The nature of this cache means that some rare
events might be missed. To check for that, running
the code multiple times and, each time, double the
cache size. Stop when doubling the cache size stops
changing the output.

Just as an example of that process, here we are logging 1,000,000 numbers in a l
og with a cache of size 16.
Note that the resulting cache is much smaller than 1,000,000 items. Also, the co
ntents of the cache
come from the entire range one to one million (so our log is not biased to just
the first few samples:

 % python −i log.py
 >>> The.cache.keep = 16
 >>> log = Num()
 >>> for x in xrange(1000000): log += x
 >>> sorted(log._cache)
 [77748, 114712, 122521, 224268,
 289880, 313675, 502464, 625036,
 661881, 663207, 680085, 684674,
 867075, 875594, 922141, 945896]
 >>>

 ### Caching Slow Reports

 Some of the things we want to report from these logs take a little while to cal
culate (e.g. finding the median
    requires a sort of a numeric cache):

+ Such reports should be run and cached so they can be accessed many time withou
t the need
for tedious recalculation.
+ These reports become outdated if new log information arrives so the following
code deletes these reports if ever new data arrives.
+ The protocol for access those reports is to call _log.has().x_ where "x" is a
field
generated by the report.  Log subclasses generate reports using the special _rep
ort()_ method
(see examples, below).

Just as an example of reporting, after the above run (where we logged 1,000,000
numbers), the following reports are available:

>>> log.has().lo
0
>>> log.has().hi
945896
>>> print log.has().median # 50th percentile
662544.0
```

```
>>> print log.has().iqr # (75−25)th percentile
205194

Note that our median is not as expected (it should be around half a million). Wh
y? Well, clearly a cache of size 16 is
too small to track a million numbers. So how many numbers do we need? Well, that
 depends on the distribution being explored
but here’s how the median is effected by cache size for uniform distributions:

>>> for size in [16,32,64,128,256]:
...     The.cache.keep=size
...     log = Num()
...     for x in xrange(1000000): log += x
...     print size, ":" log.has().median
...
16 : 637374.5
32 : 480145.5
64 : 520585.5
128 : 490742.0
256 : 470870.5


Note that we get pretty close to half a million with cache sizes at 32 or above.
 And the lesson: sometimes, a limited
sample can offer a useful approximation to a seemingly complex process.

## Standard Header
"""
from __future__ import division, print_function
import sys, random, math, datetime, time, re
from base import memo
import base
import functools

class Log(object):
    "Keep a random sample of stuff seen so far."

    def __init__(self, inits=None, label=None, max_size=256):
        self._cache            = []
        self._n                = 0
        self._report           = None
        self.label             = label or ’’
        self.max_size          = max_size
        self._valid_statistics = False
        if inits:
            map(self.__iadd__, inits)

    def random_index(self):
        return base.random_index(self._cache)

    def __iadd__(self, x):
        if x is None:
            return x
        self._n += 1
        changed = False

        # if cache has room, add item
        if self.max_size is None or len(self._cache) < self.max_size:
            changed = True
            self._cache.append(x)
        # cache is full: maybe replace an old item
        else:
            # items less likely to be replaced later in the run:
            # leads to uniform sample of entire run
            if random.random() <= self.max_size / self._n:
                changed = True
                self._cache[self.random_index()] = x

        if changed:
            self._invalidate_statistics()
            self._change(x)

        return self
```

```

    def any(self):
        return random.choice(self._cache)

    def report(self):
        if self._report is None:
            self._report = self.generate_report()
        return self._report

    def setup(self):
        raise NotImplementedError()

    def _invalidate_statistics(self):
        ’’’
        default implementation. if _valid_statistics is something other than
        a boolean, reimplement!
        ’’’
        self._valid_statistics = False

    def ish(self, *args, **kwargs):
        raise NotImplementedError()

    def _change(self, x):
        ’’’
        override to add incremental updating functionality
        ’’’
        pass

    def _prepare_data(self):
        s = ’_prepare_data() not implemented for ’ + self.__class__.__name__
        raise NotImplementedError(s)

    @staticmethod
    def log_for(t):
        if t == int or t == float or isinstance(t, (int, float)):
            return NumberLog()
        else:
            return SymbolLog()


def statistic(f):
    ’’’
    decorator for log functions that return statistics about contents.
    if _valid_statistics is False, generate valid stats before calling
    the wrapped function.
    ’’’
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        self = args[0]
        if not self._valid_statistics:
            self._prepare_data()
        return f(*args, **kwargs)

    return wrapper


"""
### Num

A _Num_ is a _Log_ for numbers.

+ Tracks _lo_ and _hi_ values.
+ Reports median and the IQR the (75−25)th range.
+ Generates numbers from the log by a three−way interpolation (see _ish()_).


"""
class NumberLog(Log):

    def __init__(self, *args, **kwargs):
        super(NumberLog, self).__init__(*args, **kwargs)
        assert self._n == 0
        # set to values that will be immediately overridden
        self.lo, self.hi = sys.maxint, -sys.maxint

    def _change(self, x):
        # update lo,hi
        self.lo = min(self.lo, x)
        self.hi = max(self.hi, x)

    def _prepare_data(self):
        if not self._valid_statistics:
            self._cache.sort()
        self._valid_statistics = True

    def norm(self,x):
        "normalize the argument with respect to maximum and minimum"
        if self.hi == self.lo:
            raise ValueError('hi and lo of {} are equal'.format(self.__name__))
        return (x - self.lo) / (self.hi - self.lo)

    def generate_report(self):
        return memo(median=self.median(), iqr=self.iqr(),
            lo=self.lo, hi=self.hi)

    def ish(self,f=0.1):
        """return a num likely to be similar to/representative of
        nums in the distribution"""
        return self.any() + f*(self.any() - self.any())

    @statistic
    def median(self):
        n = len(self._cache)
        center = n // 2
        if n % 2:
            return self._cache[center]
        center_next = center + 1
        center_next = max(0, min(center_next, n))
        return (self._cache[center] + self._cache[center_next]) / 2

    def mean(self):
        n = len(self._cache)
        return sum(self._cache) / n

    @statistic
    def iqr(self):
        self.sort()
        n = len(self._cache)
        return self._cache[int(n*.75)] - self._cache[int(n*.5)]

    def total(self):
        return sum(self._cache)

    @statistic
    def xtile(self, lo=0, hi=0.001,
            width=50,
            chops=[0.1, 0.3, 0.5, 0.7, 0.9],
            marks=["-", " ", " ", "-", " "],
            bar="|", star="*",
            show=" {: >6.2f}"):
        """The function _xtile_ takes a list of (possibly)
        unsorted numbers and presents them as a horizontal
        xtile chart (in ascii format). The default is a
        contracted _quintile_ that shows the
        10,30,50,70,90 breaks in the data (but this can be
        changed- see the optional flags of the function).
        """

        lo = min(lo,self._cache[0])
        hi = max(hi,self._cache[-1])

        pos = lambda p: self._cache[int(len(self._cache) * p)]
        place = lambda x: min(width-1, int(width * float((x - lo))/(hi - lo)))
        pretty = lambda xs: ','.join([show.format(x) for x in xs])

```

```
        what    = [pos(p)   for p in chops]
        where   = [place(n) for n in  what]

        out     = [' '] * width

        for one,two in base.pairs(where):
            for i in range(one, two):
                out[i] = marks[0]
            marks = marks[1:]

        out[int(width / 2)]  = bar
        out[place(pos(0.5))] = star

        return ''.join(out) +  "," +  pretty(what)

"""

WARNING: the call to _sorted_ in _report()_ makes this code
a candidate for a massive CPU suck (it is always sorting newly arrived data).
So distinguish between _adding_ things to a log in the _last_ era and
using that information in the _next_ era (so the log from the last era
    is staple in the current).

### Sym

A _Sym_ is a _Log_ for non-numerics.

+ Tracks frequency counts for symbols, and the most common symbol (the _mode_);
+ Reports the entropy of the space (a measure of diversity: lower values mean fewer rarer symbols);
+ Generated symbols from the log by returning symbols at the same probability of the frequency counts (see _ish()_).

"""
class SymbolLog(Log):

    @property
    def valid_statistics(self):
        return self._counts is None

    def _invalidate_statistics(self):
        # `_counts is None` => invalidation of calculated statistics
        # _mode would be a bad idea: what's the 'null' equivalent,
        # when None is a valid index into _counts?
        self._counts = None

    def _prepare_data(self):
        counts = {}
        mode = None
        mode_count = 0

        for x in self._cache:
            c = counts[x] = counts.get(x, 0) + 1
            if c > mode_count:
                mode = x

        self._counts, self._mode = counts, mode
        return self._counts, self._mode

    @statistic
    def counts(self):
        return self._counts

    @statistic
    def mode(self):
        return self._mode

    @statistic
    def distribution(self):
        return {k: v / len(self._cache) for k, v in self.counts().items()}

    def generate_report(self):
        return memo(
            distribution = self.distribution(),
            entropy      = self.entropy(),
            mode         = self.mode())

    @statistic
    def ish(self):
        tmp = 0
        threshold = random.random()
        for k, v in self.distribution().items():
            tmp += v
            if tmp >= threshold:
                return k
        # this shouldn’t happen, but just in case...
        return random.choice(self._cache)

    @statistic
    def entropy(self,e=0):
        n = len(self._cache)
        for k, v in self.counts().items():
            p = v / n
            # TODO: understand this equation better
            e −= p * math.log(p, 2) if p else 0
        return e
```

```
from independent_variable import IndependentVariable

from schaffer import Schaffer
from kursawe import Kursawe
from fonseca import Fonseca
from zdt1 import ZDT1
from zdt3 import ZDT3
from viennet3 import Viennet3
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV
from witschey.base import memo_sqrt

class Fonseca(Model):
    def __init__(self, ivs=3):
        ivs = tuple(IV(min=−4, max=4) for _ in xrange(ivs − 1))

        def f1(xs):
            e = sum((x − (1 / memo_sqrt(i+1))) ** 2 for i, x in enumerate(xs))
            return 1 − math.exp(−e)

        def f2(xs):
            e = sum((x + (1 / memo_sqrt(i+1))) ** 2 for i, x in enumerate(xs))
            return 1 − math.exp(−e)

        super(Fonseca, self).__init__(independents=ivs, dependents=(f1, f2))
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

import random

class IndependentVariable(object):
    def __init__(self, min=None, max=None, type=float):
        self.min = min
        self.max = max
        self.type = type

    def __call__(self):
        if self.type == float:
            f = random.uniform
        elif self.type == int:
            f = random.randint

        return f(self.min, self.max)
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV

class Kursawe(Model):
    def __init__(self, ivs=3, a=0.8, b=3):
        ivs = tuple(IV(min=−5, max=5) for _ in xrange(ivs − 1))
        self.a = a
        self.b = b

        def f1(xs):
            rv = 0
            for i in xrange(len(xs) − 1):
                exponent = (−0.2) * math.sqrt(xs[i] ** 2 + xs[i+1] ** 2)
                rv += −10 * math.exp(exponent)
            return rv

        def f2(xs):
            f = lambda x: (math.fabs(x)**self.a) + (5 * math.sin(x)**self.b)
            return sum(f(x) for x in xs)

        super(Kursawe, self).__init__(independents=ivs, dependents=(f1, f2))
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

class Model(object):
    def __init__(self, independents=None, dependents=None,
        energy_min=None, energy_max=None, enforce_energy_constraints=False):
        if independents is None or dependents is None:
            raise ValueError

        self.xs = independents
        self.ys = dependents
        self.energy_max = energy_max
        self.energy_min = energy_min
        self.enforce_energy_constraints = enforce_energy_constraints

    def normalize(self, x):
        n = x − self.energy_min
        d = self.energy_max − self.energy_min
        try:
            return n / d
        except ZeroDivisionError:
            return 0.5

    def random_input_vector(self):
        return tuple(x() for x in self.xs)

    def __call__(self, v, vector=False, norm=False):
        energy_vector = tuple(y(v) for y in self.ys)
        energy_total = sum(energy_vector)

        if self.enforce_energy_constraints:
            energy_errmsg =’current energy {} not in range [{}, {}]’.format(
                energy_total, self.energy_min, self.energy_max)

        if self.energy_min is None or self.energy_min > energy_total:
            if self.enforce_energy_constraints:
                raise ValueError(energy_errmsg)
            self.energy_min = energy_total

        if self.energy_max is None or energy_total > self.energy_max:
            if self.enforce_energy_constraints:
                raise ValueError(energy_errmsg)
            self.energy_max = energy_total

        if vector:
            return energy_vector
        if norm:
            return self.normalize(energy_total)

        return energy_total
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

from model import Model
from independent_variable import IndependentVariable as IV

class Schaffer(Model):
    def __init__(self, ivs=1):
        ivs = tuple(IV(min=−10^5, max=10^5) for _ in xrange(ivs))
        # we use def instead of lambdas so the functions keep their __name__s
        def f1(xs):
            return sum(x ** 2 for x in xs)
        def f2(xs):
            return sum((x − 2) ** 2 for x in xs)

        super(Schaffer, self).__init__(
            independents=ivs,dependents=(f1, f2))
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV

class Viennet3(Model):

    def __init__(self):

        def f1(xs):
            x_1sq = xs[0] ** 2
            x_2sq = xs[1] ** 2
            a = 0.5 * x_1sq
            b = math.sin(x_1sq + x_2sq)
            return a + x_2sq + b

        def f2(xs):
            x_1 = xs[0]
            x_2 = xs[1]

            a = ((3 * x_1 − 2 * x_2 + 4) ** 2) / 8
            b = ((x_1 + x_2 + 1) ** 2) / 27

            return a + b + 15

        def f3(xs):
            x_1sq = xs[0] ** 2
            x_2sq = xs[1] ** 2

            a = 1 / (x_1sq + x_2sq + 1)
            b = 1.1 * math.exp(−x_1sq − x_2sq)

            return a − b

        ivs = (IV(min=−3, max=3), IV(min=−3, max=3))

        super(Viennet3, self).__init__(
            independents=ivs, dependents=(f1, f2, f3))
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV

class ZDT1(Model):
    def __init__(self, ivs=30):

        def g(xs):
            return 1 + 9 * sum(xs[1:]) / (len(xs) - 1)

        def f1(xs):
            return xs[0]

        def f2(xs):
            gxs = g(xs)
            return gxs * (1 - math.sqrt(xs[0] / gxs))

        ivs = tuple(IV(min=0, max=1) for _ in xrange(30))
        super(ZDT1, self).__init__(independents=ivs, dependents=(f1, f2, g))
```

```
# all adapted from Dr. Tim Menzies’ model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division

import math

from model import Model
from independent_variable import IndependentVariable as IV
from witschey.base import memo_sqrt


class ZDT3(Model):

    def __init__(self, ivs=30):

        def g(xs):
            return 1 + 9 * sum(xs[1:]) / (len(xs) - 1)

        def f1(xs):
            return xs[0]

        def f2(xs):
            gxs = g(xs)
            a = 1 - memo_sqrt(xs[0] / gxs) - (xs[0] / gxs)
            a *= math.sin(10 * math.pi * xs[0])
            return gxs * a

        ivs = tuple(IV(min=0, max=1) for _ in xrange(30))

        super(ZDT3, self).__init__(independents=ivs, dependents=(f1, f2, g))
```

```
from simulated_annealer import SimulatedAnnealer
from maxwalksat import MaxWalkSat
```

```
from __future__ import division

import random
import numpy as np
from collections import defaultdict

from searcher import Searcher
from witschey.base import memo, tuple_replace
from witschey.log import NumberLog

class MaxWalkSat(Searcher):

    def __init__(self, model, *args, **kw):
        super(MaxWalkSat, self).__init__(model=model, *args, **kw)

    def local_search_inputs(self, bottom, top, n=10):
        chunk_length = (top − bottom) / n

        for a in np.arange(bottom, top, chunk_length):
            yield random.uniform(a, a + chunk_length)


    def run(self, text_report=True):
        rv = memo(report=’’)

        if self.spec.log_eras:
            rv.era_logs = {f.__name__: defaultdict(NumberLog)
                for f in self.model.ys}

        def report(s):
            if text_report:
                rv.report += s

        init = self.model.random_input_vector()
        solution = init
        state = solution
        current_energy = self.model(state)
        solution_energy = current_energy
        evals = 0

        report(’{: .2}’.format(solution_energy) + ’ ’)

        while evals < self.spec.iterations:

            for j in range(20):
                if evals > self.spec.iterations:
                    break

                dimension = random.randint(0, len(state) − 1)
                if self.spec.p_mutation > random.random():
                    state = tuple_replace(state,
                        dimension, self.model.xs[dimension]())

                    current_energy = self.model(state)

                    if current_energy < solution_energy:
                        solution = state
                        solution_energy = current_energy
                        report(’+’)
                    else:
                        report(’.’)

                    evals += 1

                    if self.spec.log_eras:
                        era = evals // self.spec.era_length
                        for f, v in zip(self.model.ys, self.model(state, vector=
True)):
                            rv.era_logs[f.__name__][era] += v

                    if evals % self.spec.era_length == 0:
                        report(’\n{: .2}’.format(solution_energy) + ’ ’)
                else:
                    for j in self.local_search_inputs(
                        self.model.xs[dimension].min,
                        self.model.xs[dimension].max
                        ):
                        state = tuple_replace(state,
                            dimension, self.model.xs[dimension]())

                        current_energy = self.model(state)

                        if current_energy < solution_energy:
                            solution = state
                            solution_energy = current_energy
                            report(’|’)
                        else:
                            report(’.’)

                        if self.spec.log_eras:
                            era = evals // self.spec.era_length
                            for f, v in zip(self.model.ys, self.model(state, vec
tor=True)):
                                rv.era_logs[f.__name__][era] += v


                        evals += 1
                        if evals % self.spec.era_length == 0:
                            report(’\n{: .2}’.format(solution_energy) + ’ ’)

        rv.best = solution_energy
        return rv
```

```
from __future__ import division, unicode_literals

from witschey.base import memo, The

from datetime import datetime

class Searcher(object):

    def __new__(cls, *args, **kwargs):
        # construct our object
        future_self = super(Searcher, cls).__new__(cls, *args, **kwargs)

        name = cls.__name__
        # initialize a dict with searcher’s name
        # and the initialization time
        d = dict(searcher=name, initialized=datetime.now())

        # if there are global options for this class or its bases in The
        for k in [name] + [k.__name__ for k in cls.__bases__]:
            if hasattr(The, k):
                # add them to the dict
                d.update(getattr(The, k).__dict__)

        # then, add the kwargs to the constructor call to the dict.
        # NB: this happens after adding options from The, so
        #     call-specific options override the globals
        d.update(kwargs)

        # set our spec with the contents of the dict
        future_self.spec = memo(**d)

        return future_self

    def __init__(self, model, *args, **kw):
        self.model = model
```

```
from __future__ import division

import random
import math
from collections import defaultdict

from searcher import Searcher
from witschey.base import memo
from witschey.log import NumberLog


class SimulatedAnnealer(Searcher):
    def __init__(self, model, *args, **kw):
        super(SimulatedAnnealer, self).__init__(model=model, *args, **kw)

    def run(self, text_report=True):
        rv = memo(report=’’)
        if self.spec.log_eras:
            rv.era_logs = {f.__name__: defaultdict(NumberLog)
                for f in self.model.ys}
        def report_append(s):
            if text_report:
                rv.report += s

        init = self.model.random_input_vector()
        solution = init
        state = solution
        rv.best = self.model(solution)


        def p(old, new, temp):
            """
            sets the threshold we compare to to decide whether to jump

            returns e^−((new−old)/temp)
            """
            numerator = new − old

            if not 0 <= numerator <= 1:
                numerator = old − new
            try:
                exponent = numerator / temp
            except ZeroDivisionError:
                return 0
            rv = math.exp(−exponent)
            if rv > 1:
                raise ValueError(’p returning greater than one’,
                    rv, old, new, temp)
            return rv

        report_append(’{: .2}’.format(rv.best) + ’ ’)

        for k in range(self.spec.iterations):
            neighbor_candidate = self.model.random_input_vector()
            neighbor = tuple(neighbor_candidate[i]
                if random.random() < self.spec.p_mutation else v
                for i, v in enumerate(state))

            rv.best = self.model(solution)
            neighbor_energy = self.model(neighbor)
            current_energy  = self.model(state)


            if neighbor_energy < rv.best:
                solution = neighbor
                rv.best = neighbor_energy
                report_append(’!’)

            if neighbor_energy < current_energy:
                state = neighbor
                report_append(’+’)
            else:
                good_idea = p(
```

```
                    self.model.normalize(current_energy),
                    self.model.normalize(neighbor_energy),
                    k / self.spec.iterations)
                if good_idea < random.random():
                    state = neighbor
                    report_append(’?’)

            report_append(’.’)
            if self.spec.log_eras:
                era = k // self.spec.era_length
                for f, v in zip(self.model.ys, self.model(neighbor, vector=True)
):
                    rv.era_logs[f.__name__][era] += v


            if k % self.spec.era_length == 0 and k != 0:
                report_append(’\n’ + ’{: .2}’.format(rv.best) + ’ ’)

        return rv
```