```
#### Schaffer
rank ,         name ,    med   ,  iqr
------------------------------------------------
   1 ,           SA ,       2  ,     0 (*             |             ), 2.00,  2.00,  2.00,  2.01,  2.05
   1 ,           GA ,       2  ,     0 (*             |             ), 2.01,  2.07,  2.13,  2.41,  2.80
   2 ,          MWS ,       5  ,    10 (  *-----------|---          ), 2.04,  2.14,  5.97,  8.28, 35.77

#### Kursawe
rank ,         name ,    med   ,  iqr
------------------------------------------------
   1 ,           SA ,     -12  ,     1 (-*-           |             ),-13.59, -13.29, -12.96, -12.75, -12.31
   2 ,           GA ,     -10  ,     1 (   --  *------|             ),-12.29, -11.76, -10.93, -10.61, -7.46
   3 ,          MWS ,      -7  ,     4 (       ------  *  ---------- ),-10.83, -8.76, -7.86, -6.84, -2.92

#### Fonseca
rank ,         name ,    med   ,  iqr
------------------------------------------------
   1 ,           SA ,       1  ,     0 (*-            |             ), 1.00,  1.00,  1.02,  1.04,  1.07
   2 ,           GA ,       1  ,     0 (-   *-----    |             ), 1.01,  1.05,  1.13,  1.18,  1.33
   3 ,          MWS ,       1  ,     0 (      ------   *   ---------  ), 1.21,  1.39,  1.50,  1.63,  1.91

#### ZDT1
rank ,         name ,    med   ,  iqr
------------------------------------------------
   1 ,           GA ,       7  ,     1 (      *   --   |             ), 6.92,  6.99,  7.21,  7.64,  7.86
   1 ,           SA ,       7  ,     0 (       *       |             ), 7.01,  7.10,  7.32,  7.53,  7.59
   2 ,          MWS ,       9  ,     1 (               |   --  *   -  ), 8.70,  8.91,  9.16,  9.68,  9.83

#### ZDT3
rank ,         name ,    med   ,  iqr
------------------------------------------------
   1 ,           SA ,       1  ,     0 (*             |             ), 1.14,  1.16,  1.18,  1.19,  1.22
   2 ,           GA ,       1  ,     0 (*--           |             ), 1.12,  1.17,  1.29,  1.69,  2.06
   3 ,          MWS ,       5  ,     4 (    ----     * |  ---------  ), 2.56,  3.73,  5.40,  7.06,  9.70

#### Viennet3
rank ,         name ,    med   ,  iqr
------------------------------------------------
   1 ,           SA ,      16  ,     0 (*-            |             ),15.99, 16.00, 16.01, 16.07, 16.21
   2 ,           GA ,      16  ,     0 (- *  ---      |             ),16.06, 16.13, 16.22, 16.54, 16.84
   3 ,          MWS ,      17  ,     1 (    ---    *   |------------ ),16.43, 16.79, 17.18, 17.73, 19.05

#### DTLZ7
rank ,         name ,    med   ,  iqr
------------------------------------------------
   1 ,           SA ,    3310  ,   249 (*             |             ),3154.07, 3244.91, 3310.82, 3448.35, 3452.29
   1 ,           GA ,    3535  ,   382 (*             |             ),3243.65, 3380.58, 3535.23, 3630.82, 3836.60
   1 ,          MWS ,    5643  ,  3182 (*             |             ),4117.74, 4764.19, 5643.66, 7301.41, 17698.31
```

```
from __future__ import division, print_function, unicode_literals

import json, random, functools, sys, math, itertools, collections

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):
    # from https://docs.python.org/2/library/itertools.html
    a, b = itertools.tee(xs)
    next(b, None)
    for p in itertools.izip(a, b):
        yield p

class memo(object):
    '''adapted from https://github.com/timm/sbse14/wiki/basepy'''

    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def to_str(self, depth=0, indent=4, infix=': ', sep=', ', d=None):
        return '{\n' + self._to_str(
            depth=depth + 1,
            indent=indent,
            infix=infix,
            sep=sep,
            d = self.__dict__ if d is None else d) + '}'

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

        for k in after:
            rv += ''.join([' ' * depth * indent, k, infix, '{\n'])
            k = d[k]
            k = k if type(k) == dict else k.__dict__
            rv += ''.join([self._to_str(depth=depth+1,indent=indent,
                infix=infix, sep=sep, d=k),
                ' ' * depth * indent,
                '}\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, dict):
        return random.choice(x.keys)
    if isinstance(x, collections.Iterable):
        return random.randint(0, len(x) - 1)
    raise ValueError('{} is not a list, tuple or dict'.format(x))

class StringBuilder(object):
    def __init__(self, *args):
        self._s = ''.join(args)
        self._next = []

    def append(self, arg):
        'recurse through iterables in args, adding all strings to _next '
        'raises TypeError if it finds a non-Iterable non-string'
        if isinstance(arg, basestring):
            self._next.append(arg)
        elif isinstance(arg, collections.Iterable):
            map(self.append, arg)
        else:
            raise TypeError('{} not a string or iterable'.format(arg))

    def __iadd__(self, arg):
        self.append(arg)
        return self

    def as_str(self):
        'build and cache _s if necessary, then return it.'
        if self._next:
            self._s += ''.join(self._next)
            self._next = []
        return self._s

    def __repr__(self):
        return "{}('{}')".format(self.__class__.__name__, self.as_str())

class NullObject(object):
    __slots__ = ()
    def __init__(self, *args, **kw):          return None
    def __getattribute__(self, *name, **kw): return self
    def __setattr__(self, *args, **kw):       return self
    def __iadd__(self, *args, **kw):          return self
    def __call__(self, *args, **kw):          return self
    def __bool__(self, *args, **kw):          return False
    __nonzero__ = __bool__

The = memo(
    Searcher=memo(era_length=50, terminate_early=True,
        log_eras_best_energy=True, log_eras_by_objective=False,
        iterations=1000, p_mutation=1/3),
    SimulatedAnnealer=memo(cooling_factor=.8),
    MaxWalkSat=memo(),
    GeneticAlgorithm=memo(population_size=50, p_mutation=.6))
```

```
from __future__ import division, print_function

import sortedcontainers, math

def basic_stats_sorted(xs):
    return xs if isinstance(xs, sortedcontainers.SortedList) else sorted(xs)

def median(xs):
    # implementation from http://stackoverflow.com/a/10482734/3408454
    xs = basic_stats_sorted(xs)
    n = len(xs)
    return xs[n // 2] if n % 2 else (xs[n // 2] + xs[n // 2 - 1]) / 2

def mean(xs):
    print(xs)
    return sum(xs) / len(xs)

def iqr(xs):
    n = len(xs)
    return xs[int(n * .75)] - xs[int(n * .25)]

def standard_deviation(xs, mean=None):
    if mean is None: mean = mean(xs)
    return math.sqrt((sum(x - mean) for x in xs) ** 2)

def norm(x, lo, hi):
    return (x - lo) / (hi - lo)

def xtile(xs, 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).
    """

    xs = basic_stats_sorted(xs)

    lo = min(lo, xs[0])
    hi = max(hi, xs[-1])
    if hi == lo:
        hi += .001 # ugh

    out     = [' '] * width

    pos = lambda p: xs[int(len(xs) * p)]
    place  = lambda x: min(width-1, int(len(out) * norm(x, lo, hi)))

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

    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) +  "," +  ','.join([show.format(x) for x in what])
```



```
from __future__ import division, print_function

import random, functools, collections

from sortedcontainers import SortedList

from witschey import base
from witschey.base import memo

class Log(object):
    """Keep a random sample of stuff seen so far. Based on Dr. Menzies'
    implementation."""

    MAX_SIZE = 256

    def __init__(self, inits=None, label=None, max_size=MAX_SIZE):
        self._cache            = SortedList()
        self._report           = None
        self.label             = label or ''
        self._n                = 0
        self.max_size          = max_size
        self._valid_statistics = False
        self._invalidate_statistics()
        if inits:
            map(self.__iadd__, inits)

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

    @classmethod
    def wrap(cls, x, max_size=MAX_SIZE):
        if isinstance(x, cls): return x
        return cls(inits=x, max_size=max_size)

    def __len__(self):
        return len(self._cache)

    def extend(self, xs):
        if not isinstance(xs, collections.Iterable):
            raise TypeError()
        map(self.__iadd__, xs)

    def __iadd__(self, x):
        if x is None:
            return x

        self._n += 1

        if isinstance(x, Log):
            map(self.__iadd__, x._cache)

        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.add(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 / len(self):
                changed = True
                self._cache.remove(random.choice(self._cache))
                self._cache.add(x)

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

        return self

    def __add__(self, x, max_size=MAX_SIZE):
```

```
            inits = itertools.chain(self._cache, x._cache)
            return NumberLog(inits=inits, max_size=max_size)

        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 as_list(self):
            return self._cache.as_list()

        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 contents(self):
            return self._cache.as_list()


    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
```



```
    from __future__ import division

    import sys

    from log import Log
    from witschey import base
    from witschey import basic_stats

    class NumberLog(Log):

        def __init__(self, *args, **kwargs):
            super(NumberLog, self).__init__(*args, **kwargs)

            self._invalidate_statistics()

        @property
        def hi(self):
            return self._cache[-1] # assumes SortedList implementation
        @property
        def lo(self):
            return self._cache[0]  # assumes SortedList implementation

        def _invalidate_statistics(self):
            self._cached_mean, self._cached_median = None, None
            self._cached_sd, self._cached_iqr = None, None

            super(NumberLog, self)._invalidate_statistics()

        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 basic_stats.norm(x, self.lo, self.hi)

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

        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())

        def median(self):
            if self._cached_median is not None:
                return self._cached_median
            self._cached_median = basic_stats.median(self._cache)
            return self._cached_median

        def mean(self):
            if self._cached_mean is not None:
                return self._cached_mean
            self._cached_mean = basic_stats.mean(self._cache)
            return self._cached_mean

        def standard_deviation(self):
            if self._cached_sd is not None:
                return self._cached_sd
            self._cached_sd = basic_stats.standard_deviation(
                self._cache, mean=self.mean())
            return self._cached_sd

        def iqr(self):
            if self._cached_iqr is not None:
                return self._cached_iqr
            self._cached_iqr = basic_stats.iqr(self._cache)
            return self._cached_iqr
```

```
    def xtile(self, *args, **kw):
        return basic_stats.xtile(self._cache, *args, **kw)

    def better(self, log2):
        if not self._cache or not log2._cache: return False
        if self.median() < log2.median(): return True
        if self.iqr() < log2.iqr(): return True
        return False
```



```
from model import Model
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
from dtlz7 import DTLZ7

del model
```

```
from __future__ import division, print_function

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

from abc import ABCMeta, abstractmethod

class Model(object):
    # allows us to get all subclasses with __subclasses__()
    __metaclass__ = ABCMeta

    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, 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

        return energy_vector

    def energy(self, energy_vector):
        return sum(energy_vector)
```



```
# 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 DTLZ7(Model):
    def __init__(self, ivs=30, dvs=20):

        # h/t http://stackoverflow.com/a/13184536/3408454
        # dynamically generate these suckers

        generated_fs = []
        for x in xrange(1, dvs):
            f = lambda xs: xs[x]
            f.__name__ = 'f{}'.format(x)
            generated_fs.append(f)

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

        def h(xs, fs=generated_fs, g=g):
            s = 0
            for f in fs:
                fxs = f(xs)
                a = fxs / (1 + g(xs))
                b = 1 + math.sin(3 * math.pi * fxs)
                s += a * b

            return ivs - s

        def final_f(xs):
            return (1 + g(xs)) * h(xs)
        final_f.__name__ = 'f{}'.format(ivs)

        fs = tuple(generated_fs + [final_f])

        independents = tuple(IV(min=0, max=1) for _ in xrange(ivs))
        super(DTLZ7, self).__init__(independents=independents, dependents=fs)
```



```
# 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))
```

Sep 22, 14 23:13 **"csc710sbse: hw5: Witschey"** Page 1/1

```
# 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))
```

Oct 07, 14 13:17 **"csc710sbse: hw5: Witschey"** Page 1/1

```
# 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))
```

Sep 22, 14 23:13 "csc710sbse: hw5: Witschey" Page 1/1

```
# 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 __future__ import division, unicode_literals

from witschey.base import memo, The
from witschey.models import Model

from collections import OrderedDict, namedtuple
from datetime import datetime
import abc
from types import NoneType

class Searcher(object):
    # allows us to get all subclasses with __subclasses__()
    __metaclass__ = abc.ABCMeta

    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()

    def run(*args, **kwargs):
        raise NotImplementedError()

class SearcherConfig(object):

    def __init__(self, searcher=None, model=None, **kwargs):
        self.searcher, self.model = searcher, model
        self._kw_dict = kwargs

    def get_searcher(self, searcher=None, model=None, **kwargs):
        s = searcher or self.searcher
        m = model or self.model
        kw = self._kw_dict.copy().update(kwargs) or {}

        return s(m, **kw)

    @property
    def searcher(self):
        return self._searcher
    @searcher.setter
    def searcher(self, value):
        if isinstance(value, NoneType) or issubclass(value, Searcher):
            self._searcher = value
        else:
            raise TypeError('{} is not a Searcher or None'.format(value))

    @property
    def model(self):
        return self._model
    @model.setter
    def model(self, value):
```

```
            if isinstance(value, NoneType) or issubclass(value, Model):
                self._model = value
            else:
                raise TypeError('{} is not a Model or None'.format(value))

        def update(self, searcher=None, model=None, **kwargs):
            if searcher is not None: self.searcher = searcher
            if model    is not None: self.model    = model
            self._kw_dict.update(kwargs)

        def as_dict(self):
            "gives back a dict with the searcher and model first"
            return OrderedDict(searcher=self._searcher,
                model=self._model, **self._kw_dict)

        def __repr__(self):
            kw_string = ', '.join('{0}={1}'.format(k, v)
                    for k, v in self.as_dict().iteritems())
            return '{0}({1})'.format(self.__class__.__name__, kw_string)

    SearchIO = namedtuple('SearchIO', ('xs', 'ys', 'energy'))

    def compute_model_io(model, xs):
        ys = model(xs)
        return SearchIO(xs, ys, model.energy(ys))
```



```
    from __future__ import division, print_function

    import random, math
    from collections import defaultdict
    from copy import deepcopy

    from searcher import Searcher, SearchIO, compute_model_io
    from witschey.base import memo
    from witschey.log import NumberLog

    def p(old, new, temp, cooling_factor):
        """
        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 * cooling_factor


    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='')
            log_eras_by_objective =\
                self.spec.log_eras_by_objective or self.spec.terminate_early
            if log_eras_by_objective:
                rv.era_logs_by_objective = {
                    f.__name__: defaultdict(NumberLog)
                    for f in self.model.ys
                }
            if self.spec.log_eras_best_energy:
                rv.era_logs_best_energy = defaultdict(NumberLog)
            def report_append(s):
                if text_report:
                    rv.report += s

            init_xs = self.model.random_input_vector()
            init_ys = self.model(init_xs)
            best = SearchIO(init_xs, init_ys, self.model.energy(init_ys))
            current = deepcopy(best)

            report_append('{: .2}'.format(best.energy) + ' ')
            self.lives = 4

            for k in range(self.spec.iterations):
                if self.lives <= 0 and self.spec.terminate_early: break

                neighbor_candidate_xs = self.model.random_input_vector()
                neighbor_xs = tuple(current.xs[i]
                    if random.random() < self.spec.p_mutation else v
                    for i, v in enumerate(neighbor_candidate_xs))

                neighbor = compute_model_io(self.model, neighbor_candidate_xs)

                if neighbor.energy < best.energy:
                    best, current = neighbor, neighbor
                    report_append('!')

                if neighbor.energy < current.energy:
```

```
                    current = neighbor
                    report_append('+')
                else:
                    good_idea = p(
                        self.model.normalize(current.energy),
                        self.model.normalize(neighbor.energy),
                        k / self.spec.iterations, self.spec.cooling_factor)
                    # if random.random() < good_idea:
                    if good_idea < random.random():
                        current = neighbor
                        report_append('?')

                report_append('.')

                era = k // self.spec.era_length
                for f, v in zip(self.model.ys, best.ys):
                    if log_eras_by_objective:
                        rv.era_logs_by_objective[f.__name__][era] += v
                    if self.spec.log_eras_best_energy:
                        rv.era_logs_best_energy[era] += best.energy

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

                    self.lives -= 1

                    if not self.spec.terminate_early: break
                    for logs in rv.era_logs_by_objective.values():
                        if era not in logs: break
                        if len(logs.keys()) < 2: break

                        prev_log = logs[logs.keys().index(era) - 1]
                        if logs[era].better(prev_log): self.lives += 1

        rv.best = best.energy
        return rv
```



```
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='')

        log_objectives = self.spec.log_eras_by_objective or self.spec.terminate_
early
        self.lives = 4

        if log_objectives:
            rv.era_logs_by_objective = {f.__name__: defaultdict(NumberLog)
                for f in self.model.ys}
        if self.spec.log_eras_best_energy:
            rv.era_logs_best_energy = defaultdict(NumberLog)

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

        self.terminate = False
        def end_era(evals, era_length, log_value):
            report('\n{: .2}'.format(log_value) + ' ')

            self.lives -= 1
            eras = evals // era_length


            for logs in rv.era_logs_by_objective.values():
                if eras not in logs: break
                if len(logs.keys()) < 2: break

                prev_log = logs[logs.keys().index(eras) - 1]
                if logs[eras].better(prev_log): self.lives += 1

            if self.lives <= 0: self.terminate = True


        def log_era(evals, era_length, dependents_outputs):
            era = evals // era_length
            for f, v in dependents_outputs:
                if log_objectives:
                    rv.era_logs_by_objective[f.__name__][era] += v
                if self.spec.log_eras_best_energy:
                    rv.era_logs_best_energy[era] += rv.best


        init = self.model.random_input_vector()
        solution = init
        state = solution
        current_energy = self.model.energy(self.model(state))
        rv.best = current_energy
        evals = 0
```

```
        report('{: .2}'.format(rv.best) + ' ')

        while evals < self.spec.iterations:
            if self.terminate: break

            for j in range(20):
                if evals > self.spec.iterations or self.terminate:
                    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.energy(self.model(state))

                    if current_energy < rv.best:
                        solution = state
                        rv.best = current_energy
                        report('+')
                    else:
                        report('.')

                    evals += 1


                    if evals % self.spec.era_length == 0:
                        end_era(evals, self.spec.era_length, rv.best)

                else:
                    for j in self.local_search_inputs(
                        self.model.xs[dimension].min,
                        self.model.xs[dimension].max
                        ):
                        if self.terminate: break

                        state = tuple_replace(state,
                            dimension, self.model.xs[dimension]())

                        current_energy = self.model(state)

                        if current_energy < rv.best:
                            solution = state
                            rv.best = current_energy
                            report('|')
                        else:
                            report('.')

                        evals += 1
                        if evals % self.spec.era_length == 0:
                            end_era(evals, self.spec.era_length, rv.best)
                if log_objectives or self.spec.log_eras_energy:
                    log_era(evals, self.spec.era_length,
                        zip(self.model.ys, self.model(solution)))

        rv.evaluations = evals
        return rv
```



```
    from __future__ import division, print_function

    import itertools, random, collections
    from collections import defaultdict

    from witschey import base
    from witschey.base import memo
    from searcher import Searcher, SearchIO, compute_model_io
    from witschey.log import NumberLog

    # adapted from Chris Theisen's code
    #     his code provided the shell that I worked in and styled to my liking
    #Structure from:
    #http://www.cleveralgorithms.com/nature-inspired/evolution/genetic_algorithm.html

    class GeneticAlgorithm(Searcher):

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

        def mutate(self, child):
            i = base.random_index(child)
            return base.tuple_replace(child, i, self.model.xs[i]())

        def crossover(self, parent1, parent2, crossovers=1):
            if len(parent1) != len(parent2):
                raise ValueError('parents must be same length to breed')
            if len(parent1) == 1:
                return random.choice((parent1, parent2))

            if crossovers < 1:
                raise ValueError('cannot have fewer than 1 crossover')
            crossovers = min(len(parent1) - 2, crossovers)
            # print(crossovers)
            x_pts = itertools.chain((0,),
                sorted(random.sample(xrange(1, len(parent1) - 1), crossovers)),
                (None,))

            ugh_mom_dad = itertools.cycle((parent1, parent2))

            segments = [itertools.islice(parent, p[0], p[1])
                for parent, p in itertools.izip(ugh_mom_dad, base.pairs(x_pts))]

            return tuple(itertools.chain(*segments))

        def select_parents(self, population, output_size): #all possible parents
            fore = itertools.combinations(population, 2)
            back = itertools.combinations(reversed(population), 2)
            all_parents = set(fore).union(set(back))
            if len(all_parents) < output_size:
                return all_parents
            return random.sample(all_parents, output_size)

        def run(self, text_report=True):
            rand_vect = lambda: self.model.random_input_vector()
            pop_size = self.spec.population_size
            init_xs = tuple(rand_vect() for _ in xrange(pop_size))
            energy = lambda x: x.energy

            report = base.StringBuilder() if text_report else base.NullObject()
            energy_by_generation = defaultdict(
                NumberLog if self.spec.log_eras_best_energy else base.NullObject)


            population = tuple(compute_model_io(self.model, xs) for xs in init_xs)

            stop = False

            best = min(population, key=energy)

            evals = 0
            for gen in xrange(self.spec.iterations or 1000):
```

```
            children = []
            for parent1, parent2 in self.select_parents(population, pop_size):
                xs = self.crossover(parent1.xs, parent2.xs, 2)
                ys = self.model(xs)
                if random.random() < self.spec.p_mutation:
                    self.mutate(xs)
                child = SearchIO(xs, ys, self.model.energy(ys))
                children.append(child)

            best_in_pop = min(children, key=energy)

            prev_best_energy = best.energy
            best = min(best, best_in_pop, key=energy)

            report += str(best.energy)
            # passing an iterable so it isn't calculated for NullStringBuilder
            report += ('+' if x.energy < prev_best_energy else '.'
                for x in children)
            report += '\n'

            energy_by_generation[gen] += best.energy

            population = children
            evals += len(population)

            if evals > self.spec.iterations: break
            #some "is significantly better" termination logic here

        rv = memo(best=best.energy, evals=evals)
        if report: rv.report = report.as_str()
        if energy_by_generation:
            rv.era_logs_best_energy = energy_by_generation

        return rv
```



```
"""

## Hyptotheis Testing Stuff


### Standard Stuff

#### Standard Headers

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

from witschey.base import StringBuilder

"""

#### 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 Introduct
ion to the Bootstrap (Chapman & Hall/CRC Monographs on Statistics & Applied Prob
ability), 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:

365

    + 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:
370     + 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
375             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...

380

    + 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).

385

    """
    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
390   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:
395     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
400            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
405   """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):
410     "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"
415       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:
420         part.rank = rank
        return rank
      recurse(sorted(data),all)
      return data

425 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
```



```
430   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
435   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
440   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
445       if now > before:
            before,cut,left,right = now,i,l,r
      return cut,left,right

    def leftRight(parts,epsilon=0.01):
450   """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
455   """
      rights = {}
      n = j = len(parts) - 1
      while j > 0:
        rights[j] = parts[j]
460     if j < n: rights[j] += rights[j+1]
        j -=1
      left = parts[0]
      for i,one in enumerate(parts):
        if i> 0:
465       if parts[i]._median - parts[i-1]._median > epsilon:
            yield i,left,rights[i]
          left += one
    """

470 ## Putting it All Together

    Driver for the demos:

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

500 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]
        ])

    if __name__ == "__main__": rdiv8()
```