```
from __future__ import print_function, division, unicode_literals

from random import random as rand
import random
import math
import sys

try:
    model_name = sys.argv[1]
except IndexError:
    raise ValueError('needs at least 1 argument', sys.argv)

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)

class Model(object):
    def __init__(self, function,
                 input_len, input_min, input_max,
                 energy_min, energy_max,
                 iterations=1000):
        self.function = function
        self.input_max = input_max
        self.input_min = input_min
        self.energy_max = energy_max
        self.energy_min = energy_min
        self.input_len = input_len
        self.iterations = iterations

    def normalize(self, x):
        n = x - self.energy_min
        d = self.energy_max - self.energy_min
        try:
            rv = n / d
        except ZeroDivisionError:
            raise ValueError("model's max and min energy are the same!")
        return rv

    def random_input_vector(self):
        return tuple(random.uniform(self.input_min, self.input_max)
            for i in range(self.input_len))

    def __call__(self, *vals):
        energy_raw = sum(self.function(v) for v in vals)

        if not self.energy_min <= energy_raw <= self.energy_max:
            raise ValueError(
                'current energy {c} not in range [{min}, {max}]'.format(
                    c=energy_raw, min=self.energy_min, max=self.energy_max
                )
            )

        return self.normalize(energy_raw)

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
```



```
schaffer = lambda x: (x[0] ** 2) + ((x[0] - 2) ** 2)

def fonseca(t, n=3):
    assert len(t) == n
    e1, e2 = 0, 0
    for i, x in enumerate(t):
        f = math.sqrt(i + 1)
        e1 += (x - (1 / f)) ** 2
        e2 += (x + (1 / f)) ** 2

    f1 = 1 - math.exp(-e1)
    f2 = 1 - math.exp(-e2)
    return f1 + f2

def kursawe(t, n=3, a=0.8, b=3):
    assert len(t) == n

    f1 = 0
    for i in range(n - 1):
        exponent = (-0.2) * math.sqrt(t[i] ** 2 + t[i+1] ** 2)
        f1 += -10 * math.exp(exponent)

    e = lambda x: (math.fabs(x) ** a) + (5 * math.sin(x) ** b)
    f2 = sum(e(x) for x in t)

    return f1 + f2


model_table = {
    'fonseca':  Model(fonseca, 3, -4, 4, 0, 2, iterations=1500),
    'kursawe':  Model(kursawe, 3, -5, 5, -24, 21, iterations=1500)
}

try:
    model = model_table[model_name.lower()]
except KeyError as e:
    exit('{e} is an invalid model name. valid model names are {ms}'.format(
        e=e, ms=model_table.keys())
    )

init = model.random_input_vector()
solution = init
state = solution

print('Simulated Annealing: {}'.format(model_name.title()))

print(pretty_input(init) + ': ' + '{: .2f}'.format(model(solution)) + ' ',
    end='')
for k in range(model.iterations):
    neighbor_candidate = model.random_input_vector()
    neighbor = tuple([neighbor_candidate[i]
        if rand() < 0.33 else state[i]
        for i in range(len(state))
        ])

    solution_energy = model(solution)
    neighbor_energy = model(neighbor)
    current_energy  = model(state)

    if neighbor_energy < solution_energy:
        solution = neighbor
        energy_min = solution_energy
        print('!', end='')

    if neighbor_energy < current_energy:
        state = neighbor
        print('+', end='')
    elif p(current_energy, neighbor_energy, k/model.iterations) < rand():
        state = neighbor
        print('?', end='')

    print('.', end='')
```

```
        if k % 50 == 0 and k != 0:
            print('\n' + pretty_input(solution) + ': '
                + '{: .2}'.format(energy_min) + ' ', end='')
150
    print()
```



```
    from __future__ import print_function, division, unicode_literals

    from random import random as rand
    import sys
 5  import random
    import math
    import numpy as np

    def pretty_input(t, model=None):
10      float_format = lambda x: '{: .2f}'.format(x)
        str_tuple = tuple(float_format(x) for x in t)
        rv = ', '.join(s for s in str_tuple)
        if model:
            return rv + ': ' + '{: .3f}'.format(model(t))
15      return rv

    def fonseca(t, n=3):
        assert len(t) == n
        e1, e2 = 0, 0
20      for i, x in enumerate(t):
            f = math.sqrt(i + 1)
            e1 += (x - (1 / f)) ** 2
            e2 += (x + (1 / f)) ** 2

25      f1 = 1 - math.exp(-e1)
        f2 = 1 - math.exp(-e2)
        return f1 + f2

    class Model(object):
30      def __init__(self, function,
            input_len, input_min, input_max,
            energy_min, energy_max):
            self.function = function
            self.input_max = input_max
35          self.input_min = input_min
            self.energy_max = energy_max
            self.energy_min = energy_min
            self.input_len = input_len

40      def normalize(self, x):
            n = x - self.energy_min
            d = self.energy_max - self.energy_min
            try:
                rv = n / d
45          except ZeroDivisionError:
                raise ValueError("model's max and min energy are the same!")
            return rv

        def random_input(self):
50          return random.uniform(self.input_min, self.input_max)

        def random_input_vector(self):
            return tuple(self.random_input() for i in range(self.input_len))

55      def __call__(self, *vals):
            energy_raw = sum(self.function(v) for v in vals)

            if not self.energy_min <= energy_raw <= self.energy_max:
                raise ValueError(
60                  'current energy {c} not in range [{min}, {max}]'.format(
                        c=energy_raw, min=self.energy_min, max=self.energy_max
                    )
                )

65          return self.normalize(energy_raw)

    class State(object):

        def __init__(self, model, max_iterations=5000):
70          self.solution = None
            self.current  = None
            self.solution_energy = None
            self.current_energy  = None
```

```
            self.evals = 0
            self.max_iterations = max_iterations

    def local_search_inputs(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)

    model = Model(fonseca, 3, -4, 4, 0, 20)

    def maxwalksat(p=0.5):
        state = State(model)

        state.current = model.random_input_vector()
        state.solution = state.current
        state.current_energy  = model(state.current)
        state.solution_energy = model(state.solution)

        print('MaxWalkSat run, Fonseca Model\np of local search:', p)
        print(pretty_input(state.current, model=model), end=' ')

        for i in range(state.max_iterations):

            for j in range(20):
                if state.solution_energy < 0.06:
                    print('%')
                    print()
                    print('Best:', state.solution_energy)
                    return
                if state.evals > state.max_iterations:
                    print('\ntoo many iterations')
                    return

                dimension = random.randint(0, len(state.current) - 1)
                if p > rand():
                    slist = list(state.current)
                    slist[dimension] = model.random_input()
                    state.current = tuple(slist)

                    state.current_energy = model(state.current)

                    if state.current_energy < state.solution_energy:
                        state.solution = state.current
                        state.solution_energy = state.current_energy
                        print('+', end='')
                    else:
                        print('.', end='')

                    state.evals += 1
                    if state.evals % 50 == 0:
                        print('\n{}'.format(
                            pretty_input(state.current, model=model)), end=' ')

                else:
                    for i in local_search_inputs(model.input_min, model.input_max):
                        slist = list(state.current)
                        slist[dimension] = i
                        state.current = tuple(slist)

                        state.current_energy = model(state.current)

                        if state.current_energy < state.solution_energy:
                            state.solution = state.current
                            state.solution_energy = state.current_energy
                            print('|', end='')
                        else:
                            print('.', end='')

                        state.evals += 1
                        if state.evals % 50 == 0:
                            print('\n{}'.format(
                                pretty_input(state.current, model=model)),
```



```
                                end=' ')

    if __name__ == '__main__':
        maxwalksat(p=0.25)
        print()
        maxwalksat(p=0.5)
        print()
        maxwalksat(p=0.75)
```

In my test of MaxWalkSat for the Fonseca model, the algorighm converged on a value below my threshold faster when local search was more likely.
This indicates that the output of the model varies greatly along each input variable -- exploring the full space of one variable finds minima more effectively than exploring all of them simultaneously.