```
#!/usr/bin/env python

"""This script simulates the generation of English names.
It is
This work is available under http://kochanski.org/gpk/teaching/0401Oxford ,
part of the lecture titled ''Monte Carlo Simulations,''
from the Hilary Term 2004 course.
"""

# This work is licensed under the Creative Commons Attribution License.
# To view a copy of this license,
# visit http://creativecommons.org/licenses/by/1.0/
# or send a letter to Creative Commons,
# 559 Nathan Abbott Way, Stanford, California 94305, USA.
#     HISTORY
#         Written and copyright by Greg Kochanski, 2004.


import random      # Random number generators.
import Numeric     # Math on vectors and matrices.
import math        # Other maths functions.

# A list of nicknames. All are assumed to be equally probable.
Nicknames = [
                ['red'],
                ['spee','dee'],
                ['big'],
                ['push', 'ee'],
                ['blond', 'ee'],
                ['shrimp'],
                ['stink', 'ee'],
                ['book', 'worm'],
                ['sai', 'lor'],
                ['scot'],
                ['black']
                ]


# A list of occupational names:
Occ = [
                ['smith'], ['butch', 'er'], ['farm', 'er'],
                ['cow', 'man'], ['weav', 'er'],
                ['tai', 'lor'], ['tan', 'er'],
                ['brew', 'er'], ['vel', 'lum', 'mak', 'er'],
                ['car', 'pen', 'ter'], ['groom'],
                ['far', 'rier'], ['black', 'smith'],
                ['bar', 'ber'], ['gold', 'smith'],
                ['arch', 'er'], ['cook'], ['ba', 'ker'],
                ['cow', 'herd'], ['shep', 'herd'],
                ['goat', 'herd'], ['fal', 'con', 'er'],
                ['scho', 'lar'], ['mas', 'ter'],
                ['por', 'ter']
                ]
```

```
# A list of place names:
Place = [
        ['ox', 'ford'], ['hink', 'sey'],
        ['wy', 'tham'], ['thame'],
        ['wynch', 'wood'], ['bot', 'ley'],
        ['sum', 'mer', 'town'],
        ['lon', 'don'], ['york'],
        ['ches', 'ter'], ['read', 'ing'],
        ['bath'], ['ave', 'bur', 'ry'],
        ['dor', 'ches', 'ter'], ['mar', 'ston'],
        ['hea', 'ding', 'ton'], ['cow', 'ley'],
        ['cum', 'nor'], ['kid', 'ling', 'ton'],
        ['saint', 'giles'],
        ['ab', 'ing', 'don']
    ]


class nprms:
    __doc__ = """This class contains the adjustable parameters
            for the family of models."""

    def __init__(self, prms=None):
        """This function creates an instance of the class."""
        if prms is None:
            self.p1 = 0.25
            self.p2 = 0.25
            self.p3 = 0.25
            self.pdup = 5.0
        else:
            self.p1, self.p2, self.p3, self.pdup = prms

    def not_ok(self):
        """This function tests if the adjustable parameters are silly or not."""
        if self.p1<0 or self.p1>1:
            return 'p1'
        if self.p2<0 or self.p2>1:
            return 'p2'
        if self.p3<0 or self.p3>1:
            return 'p3'
        if self.pdup<0:
            return 'pdup'

def xp(old, new, operation):
    """Print the individual operations that transform one name into a new one."""
    print old, '--(%s)-->'%operation, new


class nclass:
    __doc__ = """This class represents a single name and the processes
            that transform names."""

    def __init__(self, syllablelist, nprm):
        """Create a name from a list of its syllables (syllablelist)
            and the adjustable parameters (nprm)."""
```

```
        self.sl = syllablelist
        self.np = nprm

    def __str__(self):
        """Represent the class as a string."""
        return '-'.join(self.sl)

    __repr__ = __str__


    def p1(self):
        """Prepend a nickname: process 1."""
        o = nclass(random.choice(Nicknames) + self.sl, self.np)
        # xp(self, o, 'prepend')
        return o

    def p2(self):
        """Append a placename or occupation. Process 2."""
        o = nclass(self.sl + random.choice(PlaceOcc), self.np)
        # xp(self, o, 'append')
        return o


    def p3(self):
        """Drop syllables. Process 3."""
        ns = len(self.sl)
        if ns <= 2:
            # If the name is already short, just return a copy.
            return nclass(self.sl, self.np)

        while 1:
            # Try to delete a range of syllables, and see if
            # it leaves at least two syllables.
            dropstart = random.randint(0, ns-1)
            dropend = random.randint(1, ns-1)
            if dropstart <= dropend and dropstart + (ns-dropend) >= 2:
                break # Yes! An acceptable drop.
        o = nclass(self.sl[:dropstart]+self.sl[dropend:], self.np)
        # xp(self, o, 'drop')
        return o


    def newname(self, namedict):
        """This generates the next generation's form of the
        current name."""

        # print 'NN---:', self.sl
        while 1:
            x = random.random()
            tmp = nclass(self.sl, self.np)
            if x < self.np.p1:
                tmp = tmp.p1()
            if x < self.np.p2:
                tmp = tmp.p2()
```

```
            if x < self.np.p3 * len(self.sl):
                tmp = tmp.p3()

            # Check to see if the new name duplicates other names already
            # out in the population. If so, how many? Also, does
            # it duplicate a place or occupational name?
            dups = namedict.get(str(tmp), 0) + 100000*Placedict.get(str(tmp), 0)
            if random.random() > self.np.pdup * float(dups)/float(len(namedict)):
                break # Good enough!
            # print '——–TOO COMMON'
        return tmp


    def __cmp__(self, other):
        """Compare two names."""
        return cmp(self.sl, other.sl)


def generation(namelist):
    """Computes the names in generation N+1 from the array that is passed
    into it (generation N)."""

    namedict = {}
    for t in namelist:
        namedict[str(t)] = namedict.get(str(t), 0) + 1
    N = len(namelist)
    nnl = [ ]
    for i in range(N):
        # We randomly choose names to breed names in the next generation.
        # Some will have no descendents; some will have more than one.
        parent = random.choice(namelist)
        nn = parent.newname(namedict)
        nnl.append(nn)
    return nnl


def run(N, prms=None):
    """This function runs and prints 20 generations of statistics."""
    np = nprms(prms)
    namelist = [ ]
    for i in range(N):
        namelist.append( nclass(random.choice(Nicknames+PlaceOcc), np) )
    for t in range(20):
        print '# ----GENERATION------', t
        namelist = generation(namelist)
        lenhist = {}
        smith = 0
        for n in namelist:
            ln = len(n.sl)
            lenhist[ln] = lenhist.get(ln, 0) + 1
            smith += 'smith' in n.sl
        for l in range(0,10):
            print '#LEN:', '%.3f' % (lenhist.get(l, 0)/float(N))
        print '#SMITH:', smith/float(N)
```

```
    # Print the most common names in the final generation:
    hist = {}
    for n in namelist:
        sn = str(n)
        hist[sn] = hist.get(sn, 0) + 1
    histlist = [ (v, k) for (k, v) in hist.items() ]
    histlist.sort()
    histlist.reverse()
    for (v, k) in histlist:
        if v > 1:
            print k, v


def resid(x, N):
    """This function is used to find the best-fit values
    of the adjustable parameters."""

    print 'prms=', x
    np = nprms(x)
    if np.not_ok():
        return None

    namelist = []
    NNPO = Nicknames + PlaceOcc
    for i in range(N):
        namelist.append( nclass(random.choice(NNPO), np) )
    for t in range(20):
        namelist = generation(namelist)

    lenhist = {}
    smith = 0
    for n in namelist:
        ln = len(n.sl)
        lenhist[ln] = lenhist.get(ln, 0) + 1
        smith += 'smith' in n.sl
    datasmith = 0.006
    data = [None, 0, 0.036, 0.20, 0.46, 0.23, 0.05, 0.009, 0.002, 0.0002]
    o = [ math.log((smith/float(N))/datasmith) ]
    for l in range(2,10):
        o.append(math.log( (lenhist.get(l, 0)/float(N)) / data[l]))

    hist = {}
    for n in namelist:
        sn = str(n)
        hist[sn] = hist.get(sn, 0) + 1
    histlist = [ (v, k) for (k, v) in hist.items() ]
    histlist.sort()
    histlist.reverse()
    nfd = open('names.txt', 'w')
    nfd.writelines('# x= %s\n' % str(x))
    for (v, k) in histlist:
        if v > 1:
            nfd.writelines('%s %d\n' % ( k, v) )
```

resid

```
    print 'r=', o
    return [10*r for r in o]


def start(arglist):
    """Sets the starting position for the search to find
    the best-fit adjustable parameters."""
    return Numeric.array([0.25, 0.25, 0.25, 2], Numeric.Float)

def V(start):
    """Sets the initial region over which to search for the
    the best-fit adjustable parameters."""
    return Numeric.array([[1, 0, 0, 0], [0, 0.1, 0, 0], [0, 0, 1, 0],
                          [0, 0, 0, 1]], Numeric.Float)

NI = 1000      # Used for finding best-fit adjustable parameters.
c = 10000      # Used for finding best-fit adjustable parameters.

# Next, we compute a few things that will speed up the computation.
# First, a dictionary of place names, to allow us to rapidly decide
# whether or not a newly generated name matches a place name:
Placedict = {}
for p in Place:
    Placedict[str(nclass(p, None))] = 1

# Second, we need an array of place or occupational names:
PlaceOcc = Place + Occ

if __name__ == '__main__':
    # Begin the computation. The first argument is the
    # size of the population; the second argument
    # (an array) are the values of the adjustable parameters
    # that we want to use.
    run(10000, [0.43172252, 0.00283817, 0.13898237, 0.28479306])
```