```
from collections import defaultdict , Counter
import itertools
import math
import random

class BayesNet(object):
  def __init__(self):
    self.variables=[]
    self.lookup={}
  def add(self,name,parentnames,cpt):
    parents=[self.lookup[name] for name in parentnames]
    var=Variable(name,cpt,parents)
    self.variables.append(var)
    self.lookup[name]=var
    return self



class Variable(object):
  def __init__(self,name,cpt,parents=()):
    self.__name__=name
    self.parents=parents
    self.cpt=CPTable(cpt,parents)
    self.domain=set(itertools.chain(*self.cpt.values()))
  def __repr__(self): return self.__name__


class Factor(dict):
    pass

class ProbDist(Factor):
    def __init__(self, mapping=(), **kwargs):
        if isinstance(mapping, float):
            mapping = {T: mapping, F: 1 - mapping}
        self.update(mapping, **kwargs)
        normalize(self)



class Evidence(dict):
    pass

class CPTable(dict):

    def __init__(self, mapping, parents=()):
        if len(parents) == 0 and not (isinstance(mapping, dict) and set(mapping.keys()) == {()}):
            mapping = {(): mapping}
        for (row, dist) in mapping.items():
            if len(parents) == 1 and not isinstance(row, tuple):
                row = (row,)
            self[row] = ProbDist(dist)



class Bool(int):
    __str__ = __repr__ = lambda self: 'T' if self else 'F'

T = Bool(True)
F = Bool(False)
```

```
def P(var, evidence={}):
    row = tuple(evidence[parent] for parent in var.parents)
    return var.cpt[row]

def normalize(dist):
    total = sum(dist.values())
    for key in dist:
        dist[key] = dist[key] / total
        assert 0 <= dist[key] <= 1
    return dist

def sample(probdist):
    r = random.random()
    c = 0.0
    for outcome in probdist:
        c += probdist[outcome]
        if r <= c:
            return outcome

def globalize(mapping):
    globals().update(mapping)
```

```
Earthquake = Variable('Earthquake', 0.95)
P(Earthquake)
```

```
{T: 0.95, F: 0.050000000000000044}
```

```
P(Earthquake)[T]
```

```
0.95
```

```
sample(P(Earthquake))
```

```
T
```

```
Counter(sample(P(Earthquake)) for i in range(100000))
```

```
Counter({T: 94997, F: 5003})
```

```
assert ProbDist(0.75) == ProbDist({T: 0.75, F: 0.25})
assert ProbDist(win=15, lose=3, tie=2) == ProbDist({'win': 15, 'lose': 3, 'tie': 2})
ProbDist(win=15, lose=3, tie=2)
```

```
{'win': 0.75, 'lose': 0.15, 'tie': 0.1}
```

```
Factor(a=1, b=2, c=3, d=4)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

```
ProbDist(a=1, b=2, c=3, d=4)
```

```
{'a': 0.1, 'b': 0.2, 'c': 0.3, 'd': 0.4}
```

Bayes Net from the Diagram

```
alarm_net = (BayesNet()
    .add('Burglary', [], 0.001)
    .add('Earthquake', [], 0.002)
    .add('Alarm', ['Burglary', 'Earthquake'], {(T, T): 0.95, (T, F): 0.94, (F, T): 0.29, (F, F): 0.001})
    .add('JohnCalls', ['Alarm'], {T: 0.90, F: 0.05})
    .add('MaryCalls', ['Alarm'], {T: 0.70, F: 0.01}))
globalize(alarm_net.lookup)
alarm_net.variables
```

```
[Burglary, Earthquake, Alarm, JohnCalls, MaryCalls]
```

```
P(Burglary)
```

```
{T: 0.001, F: 0.999}
```

```
P(Alarm, {Burglary: T, Earthquake: F})
```

```
{T: 0.94, F: 0.06000000000000005}
```

```
Alarm.cpt
```

```
{(T, T): {T: 0.95, F: 0.050000000000000044},
 (T, F): {T: 0.94, F: 0.06000000000000005},
 (F, T): {T: 0.29, F: 0.71},
 (F, F): {T: 0.001, F: 0.999}}
```

```
def joint_distribution(net):
    return ProbDist({row: prod(P_xi_given_parents(var, row, net)
                               for var in net.variables)
                     for row in all_rows(net)})

def all_rows(net): return itertools.product(*[var.domain for var in net.variables])

def P_xi_given_parents(var, row, net):
    dist = P(var, Evidence(zip(net.variables, row)))
    xi = row[net.variables.index(var)]
    return dist[xi]

def prod(numbers):
    result = 1
    for x in numbers:
        result *= x
    return result
```

```
set(all_rows(alarm_net))
```

```
{(F, F, F, F, F),
 (F, F, F, F, T),
 (F, F, F, T, F),
 (F, F, F, T, T),
 (F, F, T, F, F),
 (F, F, T, F, T),
 (F, F, T, T, F),
 (F, F, T, T, T),
 (F, T, F, F, F),
 (F, T, F, F, T),
 (F, T, F, T, F),
 (F, T, F, T, T),
 (F, T, T, F, F),
 (F, T, T, F, T),
 (F, T, T, T, F),
 (F, T, T, T, T),
 (T, F, F, F, F),
 (T, F, F, F, T),
 (T, F, F, T, F),
 (T, F, F, T, T),
 (T, F, T, F, F),
 (T, F, T, F, T),
 (T, F, T, T, F),
 (T, F, T, T, T),
 (T, T, F, F, F),
 (T, T, F, F, T),
 (T, T, F, T, F),
 (T, T, F, T, T),
 (T, T, T, F, F),
 (T, T, T, F, T),
 (T, T, T, T, F),
 (T, T, T, T, T)}
```

Monty Hall

```
import random

def run_trial(switch_doors, ndoors=3):
    chosen_door = random.randint(1, ndoors)
    if switch_doors:
        revealed_door = 3 if chosen_door==2 else 2
        available_doors = [dnum for dnum in range(1,ndoors+1)
                                if dnum not in (chosen_door, revealed_door)]
        chosen_door = random.choice(available_doors)
    return chosen_door == 1





def run_trials(ntrials, switch_doors, ndoors=3):

    nwins = 0
    for i in range(ntrials):
        if run_trial(switch_doors, ndoors):
            nwins += 1
    return nwins

ndoors, ntrials = 3, 10000
nwins_without_switch = run_trials(ntrials, False, ndoors)
nwins_with_switch = run_trials(ntrials, True, ndoors)

print('Monty Hall Problem with {} doors'.format(ndoors))
print('Proportion of wins without switching: {:.4f}'
            .format(nwins_without_switch/ntrials))
print('Proportion of wins with switching: {:.4f}'
            .format(nwins_with_switch/ntrials))
```

```
Monty Hall Problem with 3 doors
Proportion of wins without switching: 0.3361
Proportion of wins with switching: 0.6661
```